Deep Learning on Embedded Systems

Deep Learning on Embedded Systems

A Hands-On Approach Using Jetson Nano and Raspberry Pi

By Tariq M. Arif

WILEY

Copyright © 2025 by John Wiley & Sons Inc. All rights reserved, including rights for text and data mining and training of artificial intelligence technologies or similar technologies.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permission.

The manufacturer's authorized representative according to the EU General Product Safety Regulation is Wiley-VCH GmbH, Boschstr. 12, 69469 Weinheim, Germany, e-mail: Product_Safety@wiley.com.

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data

Names: Arif, Tariq M., author. | John Wiley & Sons, publisher.
Title: Deep learning on embedded systems : a hands-on approach using Jetson Nano and Raspberry Pi / by Tariq M. Arif.
Description: Hoboken, New Jersey : Wiley, [2025] | Includes bibliographical references and index. | Summary: "Preface The field of Artificial intelligence has undergone a significant transformation in the last decade, moving from traditional machine learning approaches to more sophisticated deep learning techniques. This evolution has brought extraordinary advancements across various industries, including healthcare, finance, transportation, manufacturing, robotics, and consumer technology. For this reason, there is a growing need to incorporate deep learning technology in various research projects and academic curricula. As customizable embedded devices become more affordable and portable for deploying AI models, the growing demand for exploring this technology is also spreading across all age groups, from children to the elderly. This book aims to address this demand and serves as a comprehensive hands-on guide to understanding the integration of deep learning with modern embedded systems, such as Jetson Nano and Raspberry Pi. It also focuses on the key components of deep learning models in simple terms without diving deeply into the statistical or mathematical theories behind them. A basic understanding of Python programming is necessary to follow the examples, as all the programs in this book are written in Python. The book introduces key concepts of deep learning and its architectures in chapters 2 and 3. Chapter 4 includes the configuration of the Windows PC used for setting up PyTorch and its related packages. This chapter also explains basic tensor operations using PyTorch. Chapter 5 and Chapter 13 include Jetson Nano and Raspberry Pi 5 configurations, respectively, along with the list of peripherals used for deploying deep learning models. As the operation of Jetson Nano and Raspberry Pi 5 involves using Linux terminals, Chapter 6 covers basic Linux terminal commands, focusing on file management and permissions. This chapter will be beneficial for readers who are unfamiliar with the Linux operating system. Chapter 7 presents the fundamentals of setting up the Docker engines and building Docker images, and demonstrates how to perform model inference within Jetson's Docker container. Chapter 11 explains how to create a deep-learning dataset for image classification and object detection using bounding boxes. The dataset developed in this chapter is utilized for model training in Chapters 9 and 10. Chapter 9 outlines the process for training a classification model, while Chapter 10 demonstrates the approach for training an object detection model with image classification"– Provided by publisher.
Identifiers: LCCN 2025007090 | ISBN 9781394269266 (hardback) | ISBN 9781394269280 (adobe pdf) | ISBN 9781394269273 (epub) | ISBN 9781394269297 (ebook other)
Subjects: LCSH: Deep learning (Machine learning). | Microcomputers–Programming. | Embedded computer systems–Programming. | Image processing–Digital techniques.
Classification: LCC Q325.73 .A75 2025 | DDC 006.3/12–dc23/eng/20250319
LC record available at https://lccn.loc.gov/2025007090

Cover Design: Wiley
Cover Image: © Motion Loop/Shutterstock

Set in 9.5/12.5pt STIXTwoText by Lumina Datamatics

Contents

Preface *x*
Acknowledgment *xi*
Biography *xii*
About the Companion Website *xiii*

1 Introduction *1*
1.1 Machine Learning to Deep Learning *1*
1.2 Modern Embedded Systems *2*
1.3 Deep Transfer Learning in Embedded System *4*
1.4 Deep Learning Frameworks: An Overview *5*
1.4.1 PyTorch *5*
1.4.2 TensorFlow *6*
1.4.3 Other Major Frameworks *6*
1.5 Deep Learning and AI: Big Data and the Road Ahead *7*
1.5.1 Advances in Computational Hardware *7*
1.5.2 Adoptions in Industry and Business *8*
1.5.3 Ethical, Legal, and Privacy Concerns *8*
1.5.4 Future Challenges *9*

2 Fundamentals of Deep Learning *15*
2.1 Neural Networks Overview *15*
2.2 Basic Concepts and Terminologies *15*
2.2.1 Neurons and Layers *15*
2.2.2 Role of Activation Functions *17*
2.2.3 Learning Types *18*
2.3 Training on a Network *18*
2.3.1 Forward and Backward Propagation *18*
2.3.2 Loss Functions *19*
2.4 Gradient Descent Algorithm *19*
2.5 Weight Initialization and Regularization *21*
2.6 Hyperparameter Tuning *22*
2.6.1 Architecture-based Hyperparameters *23*
2.6.2 Training-based Hyperparameters *23*
2.7 Overview of Common Hyperparameters *23*
2.7.1 Kernel Size *23*
2.7.2 Learning Rate *23*

2.7.3 Weight Decay *24*
2.7.4 Dropout *24*
2.7.5 Batch, Iteration, and Epochs *24*
2.8 Challenges and Best Practices for Efficient Tuning *25*

3 Convolutional and Recurrent Neural Network *27*
3.1 Introduction *27*
3.2 Historical Background *27*
3.3 Convolutional Neural Networks *28*
3.3.1 CNN Framework *28*
3.3.2 Convolutional Operations *28*
3.3.3 Padding and Stride *30*
3.3.4 Pooling Layers *31*
3.3.5 Fully Connected Layers *32*
3.3.6 Output Layers *32*
3.3.7 Overall Architecture *32*
3.4 Recurrent Neural Networks *33*
3.4.1 RNN Framework *33*
3.4.2 Basic Architectures *33*
3.4.3 Backpropagation Through Time *34*
3.4.4 Long Short-term Memory Networks *35*
3.4.5 Gated Recurrent Units *35*
3.4.6 Bidirectional RNNs *35*
3.5 Applications *36*
3.6 Conclusion *37*

4 Deep Learning Using PyTorch *41*
4.1 Introduction to PyTorch *41*
4.2 Anaconda and PyTorch for Windows System *41*
4.2.1 Anaconda Installation and Environment Setup *42*
4.2.2 Setting up the PyTorch Framework *44*
4.2.2.1 Visual Studio Professional *47*
4.2.2.2 CUDA and cuDNN Installation *48*
4.3 Other Essential Packages for Deep Learning *50*
4.4 Introduction to Tensor *52*
4.4.1 Defining a Tensor *52*
4.4.2 Tensors in PyTorch *53*
4.5 Basic Torch Operations in PyTorch *53*
4.6 Gradient Calculation in PyTorch *55*
4.7 Exercise Problems *57*

5 Introduction to Jetson Nano and Setup *59*
5.1 Introduction to Jetson Embedded Devices *59*
5.1.1 Jetson Nano *60*
5.1.2 Hardware and Power Requirements *61*
5.2 Jetpack Installation *62*
5.3 Direct Setup *65*
5.3.1 Increase Root Partition Size *65*

5.3.2 Other Settings *68*
5.3.3 Wi-Fi Driver *70*
5.4 Configure Visual Studio Code on Jetson *71*
5.5 OpenCV and PyTorch in Jetson *75*
5.6 Setting up Jetson Inference *76*
5.7 OpenCV Library and Test Video Capture Functionality *79*
5.8 Conclusion *81*
5.9 Exercise Problems *81*

6 Linux Terminal Overview *85*
6.1 Introduction *85*
6.2 Basic Terminal Commands and Syntax *86*
6.3 Overview of File System *87*
6.4 Navigating Files and Directories *87*
6.5 Create, Edit, and Delete *90*
6.5.1 Copy and Move Operations *92*
6.6 Create and Execute Python Code from Terminal *93*
6.7 Common Wildcard Characters *95*
6.8 Find, View, and Get Information *96*
6.9 Permission and Ownership *99*
6.9.1 Permissions Using Octal and Symbolic Notations *100*
6.9.2 Update Permission Using Operators *101*
6.10 Install and Uninstall Packages Using "sudo" *104*
6.11 Conclusion *105*

7 Docker Engine Setup *107*
7.1 Introduction to Docker Engine *107*
7.2 Docker in Embedded Devices *107*
7.3 Jetson Inference Docker *108*
7.3.1 Download and Test Pre-trained Models *109*
7.4 Using Host Files in Docker Environment *110*
7.4.1 Create Executable Python *111*
7.4.2 Attach Host Directory to Root System and Testing *113*
7.5 Building a Docker Image *115*
7.5.1 Runtime Image for MediaPipe Ecosystem *115*
7.5.2 Install Docker Extension *116*
7.5.3 Create Docker File and Build Image *116*
7.6 Run Python Through Docker Container *118*
7.7 Exercise Problems *119*

8 Dataset Development *121*
8.1 Introduction and Requirements *121*
8.2 Types of Datasets *121*
8.3 Manual Dataset Creation *122*
8.3.1 Classification and Detection Dataset *122*
8.3.2 LabelImg Setup *123*
8.4 Automatic Image Collection Using Embedded Device *127*
8.5 Automatic Data Labeling *128*

8.6 Data Preprocessing and Cleaning *130*
8.7 Exercise Problem *131*

9 Training Model for Image Classification *133*
9.1 Problem Statement *133*
9.2 Default Configurations and Libraries *134*
9.3 Setup Data Frame Using Annotations *134*
9.4 Dataset Class and Methods *136*
9.5 Data Loader and Model Configuration *137*
9.6 Model Training *140*
9.7 Testing and Inference *141*
9.8 Fine-tuning *142*
9.8.1 Using a Different Model *144*
9.8.2 Using Different Optimizers and Schedulers *145*
9.9 Application in Embedded System *146*
9.10 Exercise Problems *147*

10 Object Detection with Classification *149*
10.1 Introduction *149*
10.2 Import Modules and Libraries *149*
10.3 Default Configurations and Random Seeds *151*
10.4 Create Data Frame and Process Labels *151*
10.5 Training and Validation of Transformers *152*
10.6 Dataset Class and Methods *153*
10.7 Data Loader and Classification Backbone *153*
10.8 Training and Validation Approach *155*
10.9 Run Multiple Epochs and Save the Best *156*
10.10 Model Inference *157*
10.10.1 Preprocessing Functions *158*
10.10.2 Inference on Test Images *160*
10.11 Multiple Object Detection with Classification *161*
10.12 Model Inference for Multiple Object Detection *163*
10.13 Conclusion *165*
10.14 Exercise Problems *167*

11 Deploy Deep Learning Models on Jetson Nano *169*
11.1 Introduction *169*
11.2 Pre-trained Models *169*
11.3 Inference on an Image File *170*
11.4 ONNX Model *172*
11.4.1 Convert PyTorch Model to ONNX *172*
11.5 Inference on Live Video Stream *173*
11.6 Conclusion *175*
11.7 Exercise Problem *176*

12 Trained PyTorch Model: From Desktop PC to Jetson Nano *177*
12.1 Introduction *177*
12.2 Model Training on a PC *177*

12.3 ONNX Model Inference *177*
12.3.1 PyTorch's.pt to ONNX Conversion *177*
12.3.2 Image Classification Using ONNX *179*
12.3.3 PyTorch's.pth to ONNX Conversion *181*
12.3.4 Object Detection Using ONNX *181*
12.4 Conclusion *186*
12.5 Exercise Problems *186*

13 Setting up Raspberry Pi *187*
13.1 Introduction to Raspberry Pi *187*
13.2 Hardware and Power Requirements *188*
13.2.1 Direct Setup *190*
13.3 Operating System Setup *190*
13.3.1 Flashing the OS Image *191*
13.3.2 Setting up OS *194*
13.4 Create Virtual Environment *197*
13.5 PyTorch and OpenCV Installation *198*
13.6 Other Essential Packages *201*
13.7 Conclusion *206*
13.8 Exercise Problem *207*

14 Deploy Deep Learning Models on Raspberry Pi *209*
14.1 Introduction *209*
14.2 Face Detection and Recognition in Video Feeds *209*
14.2.1 Face Detection Using HOG *209*
14.2.2 Face Recognition Using Transfer Learning *211*
14.3 Real-time Object Detection *214*
14.3.1 Object Detection Using YoloV3 *214*
14.4 Real-time Classification *217*
14.4.1 Classification Using PyTorch's Quantized Model *217*
14.5 Real-time Segmentation *219*
14.5.1 Segmentation Using *k*-means Clustering *219*
14.6 Exercise Problems *221*

15 Trained PyTorch Model: From Desktop PC to Raspberry Pi *225*
15.1 Introduction *225*
15.2 Model Training on a Desktop PC *225*
15.3 PyTorch's.pth to ONNX *225*
15.3.1 ONNX Runtime *225*
15.3.2 Model Conversion in Raspberry Pi 5 *226*
15.4 ONNX Model Inference on Raspberry Pi 5 *228*
15.4.1 Import Libraries and Define Configurations *228*
15.4.2 Image Preprocessing *228*
15.4.3 Model Inference with Bounding Boxes *229*
15.5 Conclusion *231*
15.6 Exercise Problems *232*

Index *235*

Preface

The field of artificial intelligence has undergone a significant transformation in the last decade, moving from traditional machine learning approaches to more sophisticated deep learning techniques. This evolution has brought extraordinary advancements across various industries, including healthcare, finance, transportation, manufacturing, robotics, and consumer technology. For this reason, there is a growing need to incorporate deep learning technology in various research projects and academic curricula. As customizable embedded devices become more affordable and portable for deploying AI models, the growing demand for exploring this technology is also spreading across all age groups, from children to the elderly. This book aims to address this demand and serves as a comprehensive hands-on guide to understanding the integration of deep learning with modern embedded systems, such as Jetson Nano and Raspberry Pi. It also focuses on the key components of deep learning models in simple terms without diving deeply into the statistical or mathematical theories behind them.

A basic understanding of Python programming is necessary to follow the examples, as all the programs in this book are written in Python. The book introduces key concepts of deep learning and its architectures in Chapters 2 and 3. Chapter 4 includes the configuration of the Windows PC used for setting up PyTorch and its related packages. This chapter also explains basic tensor operations using PyTorch. Chapters 5 and 13 include Jetson Nano and Raspberry Pi 5 configurations, respectively, along with the list of peripherals used for deploying deep learning models. As the operation of the Jetson Nano and Raspberry Pi 5 involves using Linux terminals, Chapter 6 covers basic Linux terminal commands, focusing on file management and permissions. This chapter will be beneficial for readers who are unfamiliar with the Linux operating system. Chapter 7 presents the fundamentals of setting up the docker engines and building docker images and demonstrates how to perform model inference within Jetson's docker container. Chapter 8 explains how to create a deep learning dataset for image classification and object detection using bounding boxes. The dataset developed in this chapter is utilized for model training in Chapters 9 and 10. Chapter 9 outlines the process for training a classification model, while Chapter 10 demonstrates the approach for training an object detection model with image classification.

A common challenge in deploying deep learning models on embedded systems is the setup process, especially when handling memory-intensive models. These devices often lack the memory capacity needed to train large deep learning models. To address this, in Chapters 12 and 15, the book introduces a method where models trained on a desktop computer can be transferred to Jetson Nano and Raspberry Pi for inference. This technique can be very useful for students working on science and engineering projects that involve deploying AI models on embedded devices.

Although there is abundant information available online, integrating deep learning models into design or research projects remains a challenging task. This book aims to be a practical guide for learning applied deep learning on embedded systems using Python. We encourage readers to follow the programming steps to reproduce the results demonstrated in the book before trying out the exercises. The hands-on approach presented here should provide a comprehensive understanding of the workflow, along with the related concepts and techniques.

Acknowledgment

I would like to express sincere thanks to Dr. Md Adilur Rahim from Louisiana State University for his contributions to earlier books, which laid the foundation for writing this one. I would also like to thank Devin Bigelow and Dalton Newbrough for their help in creating the datasets.

I need to appreciate my daughters, Nuha and Nesa, for their indirect contributions that made my work easy, especially when they thoughtfully chose not to interfere with my computer and toy-looking devices. Lastly, I owe special thanks to my wife, Mahbuba Sultana, whose patience and unwavering support made this work possible.

Biography

Tariq M. Arif is an associate professor in the Department of Mechanical Engineering at Weber State University, Utah. Before this role, he served as a lecturer faculty at the University of Wisconsin, Platteville. He earned his PhD in Mechanical Engineering from the New Jersey Institute of Technology (NJIT) in 2017. His primary research interests include artificial intelligence and genetic algorithms for robotics control, computer vision, and biomedical simulations involving machine learning algorithms. He obtained his master's degree in 2011 from the University of Tokushima, Japan, and his BSc in 2005 from the Bangladesh University of Engineering and Technology (BUET). Tariq also worked in the Japanese automobile industry as a CAD/CAE engineer after completing his BSc degree. Throughout his industrial and academic career, he has participated in numerous research projects and authored several books on deep learning and its engineering applications. Currently, Tariq is working on the implementation of deep learning models for various computer vision-based controls and robotics applications.

About the Companion Website

This book is accompanied by a companion website:

www.wiley.com/go/Arif/Deep

This website includes:

- Instructor
- Student

1

Introduction

1.1 Machine Learning to Deep Learning

Machine learning is one of the artificial intelligence (AI) tools that emphasizes developing computer algorithms and statistical models to learn from data. It originated from computational learning theory and pattern recognition within AI. Machine learning algorithms create mathematical models using sample data (training data) to perform tasks without being explicitly programmed. To model complex patterns and relationships from training data, one can use neural networks, which consist of interconnected layers of nodes or neurons. This process of learning using neural networks emulates the information processing methods of human brains, which is why it is also referred to as artificial intelligence or AI. Over the last two decades, as the training data volume and complexity increased, traditional machine learning methods such as decision trees and support vector machines struggled with large-scale unstructured data such as images and text. This limitation led to the emergence of deep learning, a more advanced form of machine learning that utilizes many hidden layers in a neural network (typically 5–100) to analyze vast amounts of data and extract features. This sophisticated deep neural network approach, inspired by the human brain, has recently seen tremendous progress and accomplished significant milestones that would have been deemed impossible just a few decades ago.

The growth in deep learning technology can be attributed to several key factors: the exponential increase in computational power through the use of graphical processing units (GPUs) and tensor processing units (TPUs), advancements in neural network architectures, and the development of user-friendly, open-source deep learning frameworks such as TensorFlow and PyTorch. This field has become more accessible to researchers and practitioners, and substantial investments in this field from both academia and industry are helping continuous innovations in similar fields such as computer vision, natural language processing, and autonomous systems. As deep learning is reaching a more advanced stage of development, various scientific and engineering disciplines are recognizing its practical use cases. The emerging fields of robotics, AI, finance, environmental science, bioinformatics, genetics, and healthcare are increasingly adopting deep learning models, acknowledging their potential to drive significant advancements in their respective domains. This growing integration across diverse areas of study highlights the versatility and far-reaching impact of deep learning technology.

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

1.2 Modern Embedded Systems

In the last few years, the accessibility of powerful electronic processors has led to a surge in the incorporation of deep learning models into smaller devices. Also, as the electronic devices required for edge computing are getting cheaper and more accessible, there is a growing interest in using AI applications in embedded systems such as NVIDIA Jetson devices, Google Coral, Luxonis OAK-D cameras, Intel Movidius, etc. [1–4]. Deep learning enables these systems to perform complex tasks such as image and speech recognition, natural language processing, and anomaly detection, which were previously limited to larger, more powerful systems. Nowadays, many other embedded devices can process data locally and make real-time predictions and decisions, which is essential for autonomous driving, Internet of Things (IoT), robotics, and smart home devices.

Jetson's embedded device is utilized for a wide range of applications in industry due to its powerful AI processing capabilities in a compact space. In manufacturing, it is commonly used in computer vision tasks, such as quality control and defect detection, which can enable real-time production line monitoring and analysis [5–7]. In robotics, it supports autonomous navigation and manipulation, which is crucial for improving the efficiency and accuracy of automated systems in warehouses and logistics [8, 9]. The Jetson devices play a wide variety of roles in IoT applications, including smart city infrastructure, where it helps manage traffic flow, monitor environmental conditions, and enhance public safety [10–13]. It is also employed in numerous other areas, such as retail for customer behavior analysis and inventory management and agriculture for crop monitoring and precision farming [7, 14–16]. In academia, embedded devices such as the Jetson Nano serve as a key tool in research and development, providing practical platforms for AI, machine learning, computer vision, and robotics experiments. Recently, these devices have been continuously integrated into AI courses, robotics and mechatronics programs, and IoT based lab applications, offering hands-on experience and enabling student capstone projects [17, 18]. They are also used extensively in hackathons and robotics competitions, fostering innovation and interdisciplinary collaborations. Overall, the combination of deep learning and embedded systems opens up numerous opportunities across various science and engineering domains by making advanced AI-driven functionalities widely accessible.

Currently, several embedded AI devices are utilized in industry and academia due to their powerful processing power, compact design, and real-time data analysis capabilities. These include the NVIDIA Jetson series, such as Jetson Nano, Jetson Xavier NX, and Jetson AGX Xavier, which are popular for computer vision, robotics, and IoT applications [19]. Google Coral devices, such as the Coral Dev Board and Coral USB Accelerator, are popular for prototyping and deploying machine learning models [20]. The Intel Neural Compute Stick 2 (discontinued after 2023) is known for AI capabilities in IoT devices, and the Raspberry Pi, often paired with AI accelerators such as the Coral USB Accelerator, is widely used in educational labs, projects, and small-scale AI experiments [21, 22]. A list of common embedded devices that can be utilized for small-scale AI applications and deep learning inference is summarized in Table 1.1.

In addition to these commonly used embedded devices, various other boards are capable of performing AI operations. For example, Texas Instrument's Edge AI processor TI TDA4VM is utilized in automotive and industrial applications, and ARM Cortex-M Series processors, such as the Cortex-M7, are suitable for IoT sensors and low-power devices [23, 24]. Sony's Spresense board can

Table 1.1 Processing power and memory of common embedded devices.

Embedded device	Configuration: AI performance, processing power, and memory
NVIDIA Jetson series	
Jetson Nano	472 GFLOPs, Advanced RISC Machine (ARM) Cortex-A57 CPU, and a 128-core Maxwell GPU 2 GB or 4 GB LPDDR4 memory (https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/)
Jetson TX2	1.33 TFLOPs, Dual-core Denver 2 + Quad-core ARM Cortex-A57 CPU and 256-core Pascal GPU 32 GB eMMC 5.1 memory (https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/)
Jetson Xavier NX	21 TOPs, 6-core NVIDIA Carmel ARM v8.2 64-bit CPU, and 384-core Volta with 48 Tensor Cores GPU 8 GB or 16 GB LPDDR4x memory (https://www.arrow.com/en/products/900-83668-0000-000/nvidia)
Jetson AGX Xavier	32 TOPs, 8-core ARM v8.2 64-bit CPU, 8 MB L2 + 4 MB L3, and 512-core Volta with 64 Tensor Cores GPU 32 GB LPDDR4x memory and 32 GB eMMC 5.1 storage (https://www.arrow.com/en/products/900-83668-0000-000/nvidia)
Other Jetson devices	Other variations of Jetson devices are available. A comprehensive list of specifications for these edge computing devices is available on their website. (https://marketplace.nvidia.com/en-us/robotics-edge/)
Google coral devices	
Coral Dev Board	4 TOPs, Integrated GC7000 Lite Graphics NXP i.MX 8M system-on-chip (SoC) with a quad-core ARM Cortex-A53 CPU 1 or 4 GB LPDDR4 memory (https://coral.ai/products/dev-board/#tech-specs)
Coral USB accelerator	Google Edge TPU coprocessor: 4 TOPS (int8), 2 TOPS per watt It connects to an embedded system via USB 3.0 to provide AI acceleration. (https://coral.ai/products/accelerator/)
Coral Dev Board Mini	4 TOPs, IMG PowerVR GE8300 (integrated in SoC) MediaTek 8167s SoC (Quad-core Arm Cortex-A35) CPU 2 GB LPDDR3 memory and 8 GB eMMC flash memory. (https://coral.ai/products/dev-board-mini/#tech-specs)
Coral system-on-module (SoM)	4 TOPs, Integrated GC7000 Lite Graphics NXP i.MX 8M SoC (quad Cortex-A53, Cortex-M4F) CPU 1 GB LPDDR3 memory and 8 GB eMMC flash memory. (https://coral.ai/products/som/#tech-specs)
Intel Compute Stick	
Intel Neural Compute Stick 2	1 TOPs, Intel Movidius Myriad X VPU (Vision Processing Unit) 4 GB LPDDR4 Supports a wider range of AI frameworks through the OpenVINO toolkit. However, it is currently being phased out. (https://www.intel.com/content/www/us/en/products/sku/125743/intel-movidius-neural-compute-stick/specifications.html)

(Continued)

Table 1.1 (Continued)

Embedded device	Configuration: AI performance, processing power, and memory
Raspberry Pi Boards	
Raspberry Pi 5	VideoCore VII GPU, supporting OpenGL ES 3.1, Vulkan 1.2 Broadcom BCM2712 2.4 GHz quad-core 64-bit Arm Cortex-A76 CPU LPDDR4X-4267 memory (4 GB or 8 GB) Widely used in educational projects and small-scale AI experiments Can be paired with AI accelerators such as the Coral USB Accelerator. (https://www.raspberrypi.com/products/raspberry-pi-5/)
Raspberry Pi 4 Model B	Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit CPU LPDDR4 memory (1 GB, 2 GB, 4 GB, or 8 GB) Widely used in educational projects and small-scale AI experiments Can be paired with AI accelerators such as the Coral USB Accelerator. (https://www.raspberrypi.com/products/raspberry-pi-4-model-b/)

be employed in IoT, drones, and wearable devices due to its compact size and efficiency, while Xilinx Edge AI Solutions, such as the Xilinx Zynq UltraScale+ MPSoC, are used in applications that require high-performance AI inference [25, 26]. There are a few deep learning-enabled video cameras, such as OAK-D AI, developed by Luxonis and OpenCV, which have a depth camera for AI vision systems, and AWS DeepLens (currently discontinued), which is effective for educational purposes and deep learning-based prototyping [27, 28].

1.3 Deep Transfer Learning in Embedded System

Deep transfer learning is a powerful technique that leverages the knowledge gained from a pre-trained, well-established model to a new model when there is insufficient data or processing power. This method allows knowledge learned from one type of data to be applied to a different type. In recent years, deep convolutional networks have shown remarkable success in image recognition and classification tasks using transfer learning techniques. Typically, training a deep learning model from scratch requires a vast amount of training data and computational power, which is often not available in practical scenarios. Therefore, pre-trained models on large datasets, such as ImageNet, Modified National Institute of Standards and Technology (MNIST), Microsoft Common Objects in Context, and Canadian Institute For Advanced Research, have been extensively used to enhance model performance by transferring their learned knowledge to new models [29–32]. The effectiveness of a deep learning model typically improves when it is trained on a large dataset. For example, the well-known ImageNet dataset has 14197122 images, while the Tina Image dataset includes 79302017 images [33, 34]. However, we can utilize pre-trained deep learning models that are already trained on these kinds of established and well-labeled datasets to save computational power and use embedded systems. These models can be imported and slightly adjusted to meet the specific requirements of model training.

Deep transfer learning is highly beneficial for compact embedded devices such as the Jetson Nano due to their low memory, power consumption, and processing powers. By utilizing pre-trained

models, embedded devices can effectively manage computational and memory constraints, reducing the need for extensive on-device training. This approach significantly shortens deployment times and enhances model performance with limited labeled data, making it ideal for rapid prototyping and real-time applications.

Currently, embedded systems have diverse applications across healthcare, agriculture, autonomous systems, and robotics, as the inference operation can be done in low-power or battery-powered environments. Deep transfer learning offers significant practical benefits in these areas by enabling the rapid deployment and adaptation of advanced AI models on compact embedded systems. This approach also facilitates the use of cutting-edge AI technologies in various fields, ensuring that sophisticated models can be efficiently integrated and utilized in small and low-cost devices.

1.4 Deep Learning Frameworks: An Overview

In the last decade, many deep learning frameworks have been developed, used, and gained popularity. This advancement is primarily driven by the development and support from major tech companies such as Google, Microsoft, and Facebook, as well as scientific research teams from various organizations. Another key factor driving this growth is the open-source nature of these frameworks and their associated software. The expansion of deep learning is further supported by active online communities on platforms such as Stack Overflow (https://stackoverflow.com/), Stack Exchange (https://stackexchange.com/), Quora (https://www.quora.com/), Reddit (https://www.reddit.com/), ResearchGate (www.researchgate.net), GitHub Discussions (https://github.com/orgs/community/discussions), and CodeProject (www.codeproject.com), which provide valuable resources for users of these frameworks. Some of the most popular deep learning frameworks include TensorFlow by Google's Brain Team, PyTorch by Adam et al., Keras by Chollet et al., Caffe by Berkeley Artificial Intelligence Research, and Microsoft Cognitive Toolkit (CNTK) by Microsoft Research [35–39].

1.4.1 PyTorch

In this book, we have utilized the PyTorch framework to demonstrate deep learning applications. PyTorch offers several advantages as it utilizes a dynamic computation graph, which enables on-the-fly changes to deep learning architecture. PyTorch's syntax closely resembles Python code, making it easier to learn and integrate into existing Python projects. It also integrates seamlessly with Python libraries and tools such as NumPy and SciPy. Its ecosystem includes "torchvision" library for computer vision and "torchtext" library for natural language processing, and it provides efficient performance with GPU acceleration by supporting parallelism and automatic differentiation. In recent years, PyTorch has gained significant popularity in academia, offering researchers and students access to cutting-edge methodologies and deep learning innovations. This widespread adoption is making PyTorch a powerful tool for developing and deploying sophisticated deep learning models.

In Chapters 4, 5, and 13, we walk through the process of setting up PyTorch on a Windows PC with a GPU, Jetson Nano 4 GB, and Raspberry Pi 5, respectively. Additionally, Chapters 9–12 and 14–15 highlight various use cases for PyTorch in deep learning models using these devices.

1.4.2 TensorFlow

TensorFlow, developed by Google's Brain Team, is a versatile and open-source deep learning framework that has a broad spectrum of neural network architectures and sophisticated machine learning algorithms. One of the key features of TensorFlow is its scalability, which enables model deployment across various platforms, ranging from mobile devices to large-scale distributed computing systems. Also, its extensive libraries and tools, such as "TensorBoard," facilitate the visualization and development, training, and deployment of complex models. TensorFlow is not utilized in this book for deep learning applications, as we have opted to use PyTorch for all the examples to maintain consistency and facilitate easier understanding.

1.4.3 Other Major Frameworks

Many other deep learning frameworks are gaining popularity in both industry and academia. These include Theano, Deeplearning4j (DL4J), Sonnet, Chainer, MXNet, Gluon, and open neural network exchange (ONNX) [40–45]. Although a significant number of deep learning frameworks are currently available, none of them can be considered universally superior, as each has its own strengths and weaknesses. The choice of framework often depends on the specific use case. For example, TensorFlow offers extensive libraries and strong visualization capabilities but may have slower performance and limited GPU support. In contrast, PyTorch is known for its user-friendly Python interface and robust community support but lacks some of the advanced visualization tools found in TensorFlow. Table 1.2 highlights popular deep learning frameworks along with their developer or organization and key strengths.

Table 1.2 Deep learning framework, developer, and key features.

Deep learning framework	Developer/organization	Key strengths
Tensorflow	Google Brain Team	Scalability, extensive libraries, and TensorBoard visualization
PyTorch	Facebook AI Research (FAIR)	Dynamic computation graph, strong community support, and Python integration
Keras	François Chollet	High-level API, ease of use, and rapid prototyping
Caffe	Berkeley Artificial Intelligence Research	High speed and excellent for image processing capability
Microsoft CNTK	Microsoft Research	Highly efficient and supports multiple GPUs
Theano	Université de Montréal	Symbolic computation and integration with NumPy
DL4J	Eclipse Foundation	Good for Java developers and integrates with big-data tools
Sonnet	DeepMind	Modular design built on TensorFlow
Chainer	Preferred Networks	Flexible, intuitive, and define-by-run scheme
MXNet	Apache	Lightweight, scalable, and supports multiple programming languages
Gluon	AWS and Microsoft	Simple, concise, and dynamic graph creation
ONNX	Microsoft and Facebook	Interoperability between different AI frameworks

1.5 Deep Learning and AI: Big Data and the Road Ahead

In the coming years, deep learning and AI are set to revolutionize numerous sectors and drive innovations. This technology is dependent on the utilization of big data, which refers to extremely large and complex datasets that traditional data processing tools cannot handle efficiently. Big data plays a vital role in the development of deep learning algorithms and AI systems. By using deep learning algorithms with big data, a system can be trained to effectively identify complex patterns, generate predictions, and improve its decision-making capabilities. These advanced technologies would likely bring changes across diverse industries, potentially altering the way we live, work, and interact with the world around us [46]. In the biomedical industry, these technologies are expected to enhance diagnostic accuracy by predicting dosages and customizing treatment plans [47]. Currently, AI algorithms are already being employed to analyze medical imagery, enabling early detection of diseases [48, 49]. This practical application demonstrates the tangible benefits these technologies are bringing to healthcare, potentially resulting in more effective and personalized patient care. The automotive industry will see widespread leveraging of computer vision technology for autonomous navigation and real-time decision-making. Currently, companies such as Tesla and Waymo are at the forefront of applying these technologies. However, more traditional automotive companies are gradually adopting these cutting-edge solutions into their vehicles. The customer service business is set to evolve further with the ongoing improvement of AI-driven chatbots and virtual assistants such as Siri and Alexa. These technologies will continue to elevate the user experience by providing rapid and precise responses to inquiries. At the same time, big data analytics will maintain its pivotal role in driving AI innovations across industries. Access to big data can also enable many companies to conduct in-depth analyses of consumer patterns and behaviors, allowing for the refinement of marketing strategies. Currently, Amazon's sophisticated product recommendation system is already using this type of analytics to leverage vast amounts of user data and can offer personalized suggestions very effectively.

Future smart cities are set to leverage big data analytics for various urban improvement activities. Advanced urban cities will use data-driven approaches to optimize traffic flow, reduce energy consumption, and enhance public safety measures [50, 51]. In scientific research, big data will catalyze advancements in fields such as genomics, climate science, and astrophysics by implementing novel AI or deep learning models.

Overall, the combined impact of deep learning and big data is set to transform industries, enhance living standards, and open up new paths of innovation and opportunity in the near future. As these technologies progress and become more integrated, we can expect significant changes in our work, daily lives, and approaches to solving complex problems.

1.5.1 Advances in Computational Hardware

In the near future, we can expect substantial progress in computing hardware, driven by the increasing demands for processing power, memory storage, and improved functionalities. Major tech companies such as Intel, Advanced Micro Devices (AMD), and ARM are expected to make significant breakthroughs in designing processor technology and designs. These advancements will expand the boundaries of computing power and flexibility, leading to the development of more advanced and powerful computing systems. We can expect significant technological advancements in CPU and GPU performance. Memory storage technologies are also set to evolve, with

DDR5 RAM and NVMe SSDs offering enhanced data transfer rates and storage speeds. We may see more specialized hardware for AI and machine learning, similar to NVIDIA GPUs and Google TPUs that are optimized for complex deep learning tasks.

The field of quantum computing may also grow significantly as companies such as International Business Machines Corporation (IBM) and Google make strides in solving computational challenges [52]. At the same time, current advances in semiconductor manufacturing, such as the move to 5 nm and 3 nm process nodes, suggest that it will drive further miniaturization of hardware components [53, 54]. Many emerging technologies, such as neuromorphic computing, which mimics the human brain's neural structure, could revolutionize AI processing, and silicon photonics technology can dramatically improve data communication speeds and efficiency in the future [55, 56]. Overall, we expect that these advancements will boost computing performance and efficiency, reduce hardware sizes, and open up new possibilities for the next generation.

1.5.2 Adoptions in Industry and Business

Many recent studies suggest that within the next decade, AI and deep learning will be extensively implemented across various industries, and they can transform the landscape of modern industries that are dependent on computer processes and analytics. Currently, IBM Watson Health is using AI to examine medical data for disease diagnosis and personalized treatment plans [57]. Tesla is utilizing deep learning in autonomous vehicles to improve self-driving capabilities. E-commerce giant Amazon employs AI for numerous applications, including offering personalized product recommendations, optimizing supply chain management, and operating its cashier-free Amazon Go stores [58, 59]. Microsoft is providing improved customer service with AI-driven virtual assistants [60]. In the financial sector, JPMorgan Chase is utilizing AI for fraud detection, trading, and risk assessment [61]. Siemens is incorporating AI into its manufacturing processes for predictive maintenance and quality controls, and John Deere is applying AI in agriculture for precision farming and autonomous machinery [62, 63]. These examples highlight the diverse applications in which AI is being adopted, and as the technologies and computing power advance, AI adoption will expand across numerous industries and businesses.

1.5.3 Ethical, Legal, and Privacy Concerns

The growing use of AI in our daily lives brings up several ethical, legal, and privacy concerns. From an ethical standpoint, AI systems can reinforce existing biases, leading to unfair treatment of certain groups [64, 65]. These systems are often ignored as they are not humans, leading to a lack of transparency and making it challenging to understand their decisions. The rapid advancement of AI technologies often outpaces regulatory frameworks, which makes legal issues ambiguous, especially regarding compliance and intellectual property rights [66–68]. Privacy concerns are also a major issue, as training AI models requires huge amounts of data. This raises concerns regarding potential data breaches and copyrights. AI models that employ computer vision for security applications require surveillance data for training purposes, which can lead to an invasion of privacy in both public and private spaces. To address these problems, we will need transparent and accountable AI models, stronger laws, secure data storage systems, and improved privacy protections from social media companies that continuously store and use personal data. Maintaining a balance between technological progress in AI and its ethical, legal, and privacy considerations is essential for establishing public trust in this emerging technology.

1.5.4 Future Challenges

The future incorporation of AI in our everyday lives will have challenges that will demand creative and effective solutions. Ethical issues, such as biased results in facial recognition technology, must be addressed by creating more diverse training data and implementing algorithms that can mitigate unfairness. AI poses unique challenges to academic institutions that use AI to detect AI-generated texts. Many universities and professors in English-speaking countries have a class policy that prohibits the use of AI-generated texts as part of assignments. They typically use AI-based detection tools such as Turnitin, Copyleaks, GPTZero, ZeroGPT, Sapling, GPT-2 Output Detector by Hugging Face, GLTR, and other similar applications that claim to determine AI-written texts and submissions with high accuracy. Many of these company websites promote their own research findings to justify the effectiveness of their tool for AI detection. University professors using these tools may consider assignment submissions by international students as AI-generated texts without knowing that AI tools often misclassify results for non-native English speakers [69–74]. There are numerous reasons for this bias, and more research is needed to address these issues. Trained AI detection tools assess a vast number of unknown parameters based on available datasets and might give weights to certain criteria such as sentence structure, phrases, vocabulary, and the frequency of specific types of grammatical errors. Additionally, modern grammar-checking tools (which are typically allowed in academia) use AI-generated automatic suggestions to improve sentence structure and modify writing tones and clarity. Non-native English speakers are likely to select a writing tone that is significantly different than native English speakers. These personalized selections of English vocabulary and sentences in grammar checker software vary significantly among international students from different linguistic and cultural backgrounds. At present, many AI tools such as QuillBot, Wordtune, Jasper, Stealth Writer, Originality, TextCortex, BypassAI, Phrasly.ai, StealthWriter, StealthGPT, Stealth AI, etc., claim to bypass popular AI detection tools used by academic institutions. Further research is needed to investigate generative AI, AI-based detection tools, and related technologies. Also, university professors and reviewers relying on these tools should be properly trained by experts on various aspects of misclassification and the latest innovations in the field.

Data privacy and security in AI is another issue that needs to be addressed through improved algorithms and should be monitored by regulatory bodies. Software developer companies that deploy generative models need to be overseen by individuals knowledgeable about the details of this technology. The lack of transparency in many AI algorithms hinders transparency and accountability. In the future, it is important to create AI systems that can provide clear explanations for their decisions.

AI can be utilized efficiently to boost the overall productivity of technical work made by individuals or organizations [75]. Generally, integrating AI into a company's existing infrastructure involves significant investments. Therefore, the different issues and drawbacks of a model need to be evaluated critically before being deployed on a practical level. AI-driven automation will likely raise more concerns across the globe for job displacement, and there will be a greater need for workforce retaining programs for sustainable economic growth [76, 77]. Ensuring the reliability and safety of AI systems will also be a big challenge, particularly in critical applications such as genetics, defense systems, and autonomous driving, which require robust testing and validation frameworks. Addressing these challenges is crucial for the responsible and fair implementation of AI technologies in our future society.

References

1 Mittal, S. (2019). A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *Journal of Systems Architecture* 97: 428–442.

2 Biglari, A. and Tang, W. (2023). A review of embedded machine learning based on hardware, application, and sensing scheme. *Sensors (Basel, Switzerland)* 23 (4): 2131.

3 Rojas-Perez, L.O. and Martinez-Carranza, J. (2021). Towards autonomous drone racing without GPU using an OAK-D smart camera. *Sensors (Basel, Switzerland)* 21 (22): 7436.

4 Intel (2023). *Features and Benefits of PCs with Intel® Movidius™ VPU*. https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu.html.

5 Wang, D., Hwang, J., Lee, J. et al. (2023). Temperature-based state-of-charge estimation using neural networks, gradient boosting machine and a Jetson Nano device for batteries. *Energies (Basel)* 16 (6): 2639.

6 Silva, M.O., Torres, G.M., Valadao, M.D.M. et al. (2022). Action and assembly time measurement system of Industry Workers using Jetson Nano. In: *2022 IEEE International Conference on Consumer Electronics-Taiwan*, 319–320. IEEE.

7 Beyaz, A. and Saripinar, Z. (2024). Sugar beet seed classification for production quality improvement by using YOLO and NVIDIA artificial intelligence boards. *Sugar Tech : An International Journal of Sugar Crops & Related Industries*, *26*, 1–9.

8 Vijitkunsawat, W. and Chantngarm, P. (2020). Comparison of machine learning algorithm's on self-driving car navigation using Nvidia Jetson Nano. In: *2020 17th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 201–204. IEEE.

9 Cahyo, M.P.D. and Utaminingrum, F. (2022). Autonomous robot system based on room nameplate recognition using YOLOv4 method on Jetson Nano 2GB. *JOIV : International Journal on Informatics Visualization Online* 6 (1): 117–123.

10 Hizon, J.A.J., Garcia, R.G., and Rebustillo, M.R.J. (2022). Jetson Nano and Arduino-based robot for physical distancing using Yolov4 algorithm with thermal scanner. In: *2022 IEEE 18th International Colloquium on Signal Processing & Applications (CSPA)*, 353–358. IEEE.

11 Ali, S.Z. and Khadatkar, D.R. (2021). Advanced smart lock system for multi-user environment using Nvidia Jetson Nano. *Annals of the Romanian Society for Cell Biology* 25 (3): 6930–6937.

12 Philip, A.O., Jacob, A., Tejus, K. et al. (2023). Smart standalone edge IoT device for traffic volume counting in smart cities. In: *2023 2nd International Conference on Edge Computing and Applications (ICECAA)*, 1255–1259. IEEE.

13 Uddin, M.I., Alamgir, M.S., Rahman, M.M. et al. (2021). AI traffic control system based on deepstream and IoT using NVIDIA Jetson Nano. In: *2021 2nd International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST)*, 115–119. IEEE.

14 Donapati, R.R., Cheruku, R., and Kodali, P. (2023). Real-time seed detection and germination analysis in precision agriculture: a fusion model with U-Net and CNN on Jetson Nano. *IEEE Transactions on Agrifood Electronics* 1 (2): 145–155.

15 Firdiantika, I.M., Lee, S., Bhattacharyya, C. et al. (2024). EGCY-Net: an ELAN and GhostConv-Based YOLO network for stacked packages in logistic systems. *Applied Sciences* 14 (7): 2763.

16 Kamble, S. and Bartakke, P. (2023). Smart refrigerator management system using deep learning. In: *2023 Global Conference on Information Technologies and Communications (GCITC)*, 1–6. IEEE.

17 Gong, X., Wen, Y., Liu, X. et al. (2021). Development of GPU enhanced AI education platforms for K-12 schools. In: *2021 China Automation Congress (CAC)*, 5662–5666. IEEE.

18 NVIDIA Developer (2024). *Jetson for AI Education*. https://developer.nvidia.com/embedded/learn/jetson-ai-education.

19 NVIDIA Developer (2024). *Jetson Modules*. https://developer.nvidia.com/embedded/jetson-modules.

20 Coral (2024). *Prototyping Products*. https://coral.ai/products/.

21 Intel (2022). *Intel® Neural Compute Stick 2 (Intel® NCS2)*. https://www.intel.com/content/www/us/en/developer/articles/tool/neural-compute-stick.html.

22 Mathe, S.E., Kondaveeti, H.K., Vappangi, S. et al. (2024). A comprehensive review on applications of Raspberry Pi. *Computer Science Review* 52: 100636.

23 Texas Instruments (2024). *Dual Arm® Cortex®-A72 SoC and C7x DSP with Deep-Learning, Vision and Multimedia Accelerators*. https://www.ti.com/product/TDA4VM?bm-verify=AAQAAAAJ_____dQrU0iIS490IQuzgp6ynT3VSOoYyCclkO3bLbSQfiVzvw3zH2fknDR-evsxci4BvcloR3ruKFnxFpuwBNIkyfb9Nx3uEx7cnmK2CKm5YDOa6JGMe07DegPjmDiNSOEcw_WKhHyW_1dEDnN1A2iTjGg6VvBUDk8UYgj37KN5ajghSqckz3N_EA2rIg6gduhwGqwdKlI1hs3XKrLz0GCH4dOh5Zylzwd7dWVM54oLS6l7EeVljDAOooVmoeKfK6S2xdxZadl4iB13zJlCsI_jEpJGyGDUoVTllKeZGdb3wz_d2lGOxePRe4.

24 Cortex-M4 (2024). *Control and Performance for Mixed-Signal Devices*. https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4.

25 Sony (2024). *Spresense 6-Core Microcontroller Board With Ultra-Low Power Consumption*. https://developer.sony.com/spresense.

26 Limited, A.E. (2024). *ALINX AXU2CGA: AMD Zynq UltraScale+ MPSoC XCZU2CG FPGA Development Board*. https://www.xilinx.com/products/boards-and-kits/1-1ewaaez.html.

27 Luxonis (2024). *OAK Cameras – Robotic Vision Made Simple*. https://shop.luxonis.com/collections/oak-cameras-1.

28 Barr, J. (2017). *AWS DeepLens – Get Hands-On Experience with Deep Learning With Our New Video Camera*. https://aws.amazon.com/blogs/aws/deeplens/.

29 Rastogi, R., Rastogi, A.R., Upadhyay, H. et al. (2021). Knowledge extraction in digit recognition using MNIST dataset: evolution in handwriting analysis. *International Journal of Knowledge Management* 17 (4): 1–24.

30 Tong, K. and Wu, Y. (2023). Rethinking PASCAL-VOC and MS-COCO dataset for small object detection. *Journal of Visual Communication and Image Representation* 93: 103830.

31 Ayi, M. and El-Sharkawy, M. (2020). RMNv2: reduced mobilenet V2 for CIFAR10. In: *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 0287–0292. IEEE.

32 Subramanian, M.M. and Elangovan, K. (2022). KNN approach through various image equalization techniques on CIFAR10. *NeuroQuantology* 20 (22): 3317.

33 Russakovsky, O., Deng, J., Su, H. et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115 (3): 211–252.

34 Torralba, A., Fergus, R., and Freeman, W. T. (2008). 80 million tiny images: a large data set for nonparametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30 (11): 1958–1970.

35 Abadi, M., Barham, P., Chen, J. et al. (2016). TensorFlow: a system for large-scale machine learning. *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 265–283. Savannah, GA, USA: USENIX Association.

36 Paszke, A., Gross, S., Massa, F. et al. (2019). PyTorch: an imperative style, high-performance deep learning library. *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Article 721. NY, USA: Curran Associates Inc.

37 Chollet, F. (2015). *Keras: Deep Learning for Humans*. https://github.com/keras-team/keras.

38 Jia, Y., Shelhamer, E., Donahue, J. et al. (2014). Caffe: convolutional architecture for fast feature embedding. *Proceedings of the 22nd ACM International Conference on Multimedia*, 675–678. Orlando, Florida, USA: Association for Computing Machinery.

39 Seide, F. and Agarwal, A. (2016). CNTK: Microsoft's open-source deep-learning toolkit. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2135. San Francisco, CA, USA: Association for Computing Machinery.

40 Team, T., Al-Rfou, R., Alain, G. et al. (2016). Theano: a python framework for fast computation of mathematical expressions. https://www.semanticscholar.org/paper/Theano%3A-A-Python-framework-for-fast-computation-of-Al-Rfou-Alain/6b570069f14c7588e066f7138e1f21af59d62e61.

41 Gibson, A., Nicholson, C., Patterson, J. et al. (2016). Deeplearning4j: distributed, open-source deep learning for Java and Scala on Hadoop and Spark. https://deeplearning4j.konduit.ai/en-1.0.0-beta7/getting-started/about.

42 Tokui, S., Okuta, R., Akiba, T. et al. (2019). Chainer: a deep learning framework for accelerating the research cycle. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2002–2011. Anchorage, AK, USA: Association for Computing Machinery.

43 Chen, T., Li, M., Li, Y. et al. (2015). MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv*:1512.01274.

44 Dathathri, R., Gill, G., Hoang, L. et al. (2018). Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 752–768. Philadelphia, PA, USA: Association for Computing Machinery.

45 Ahmed, S., Bisht, P., Mula, R. et al. (2021). A deep learning framework for interoperable machine learning. *Proceedings of the First International Conference on AI-ML Systems*, Article 23. Bangalore, India: Association for Computing Machinery.

46 Qiu, R. (2023). Editorial: GPT revolutionizing AI applications: empowering future digital transformation. *Digital Transformation and Society* 2 (2): 101–103.

47 Altara, R., Basson, C.J., Biondi-Zoccai, G. et al. (2024). Exploring the promise and challenges of artificial intelligence in biomedical research and clinical practice. *Journal of Cardiovascular Pharmacology* 83 (5): 403–409.

48 Ghayvat, H., Awais, M., Bashir, A.K. et al. (2023). AI-enabled radiologist in the loop: novel AI-based framework to augment radiologist performance for COVID-19 chest CT medical image annotation and classification from pneumonia. *Neural Computing and Applications* 35 (20): 14591–14609.

49 Usmani, U.A. and Usmani, M.U. (2023). AI-Driven biomedical and health informatics: Harnessing artificial intelligence for improved healthcare solutions. In: *2023 2nd International Conference on Futuristic Technologies (INCOFT)* 1–7. IEEE.

50 Nastjuk, I., Trang, S., and Papageorgiou, E.I. (2022). Smart cities and smart governance models for future cities: current research and future directions. *Electronic Markets* 32 (4): 1917–1924.

51 Javed, A.R., Shahzad, F., Rehman, S.U. et al. (2022). Future smart cities: requirements, emerging technologies, applications, challenges, and future aspects. *Cities* 129: 103794.

52 Castelvecchi, D. (2024). The AI-quantum computing mash-up: will it revolutionize science? *Nature (London)*. https://www.nature.com/articles/d41586-023-04007-0.

53 Hoeren, T. (2016). The semiconductor chip industry – The history, present and future of its IP law framework. *IIC - International Review of Intellectual Property and Competition Law* 47 (7): 763–796.

54 TSMC (2022). *3nm Technology*. https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_3nm.

55 Ran, Y., Pei, Y., Zhou, Z. et al. (2023). A review of Mott insulator in memristors: the materials, characteristics, applications for future computing systems and neuromorphic computing. *Nano Research* 16 (1): 1165–1182.

56 Sabella, R. (2020). Silicon Photonics for 5G and Future Networks. *IEEE Journal of Selected Topics in Quantum Electronics* 26 (2): 1–11.

57 Stephens, K. (2021). IBM Watson Health introduces technology for imaging AI adoption. *AXIS Imaging News*, ProQuest. Web. 21 December 2024. https://www.proquest.com/docview/2607444434/abstract/?accountid=14940&pq-origsite=summon&sourcetype=Scholarly%20Journals.

58 Amazon Announces General Availability of Amazon Q AI-Powered Assistant (2024). *The Fly*, 30 April 2024. Gale General OneFile https://link.gale.com/apps/doc/A792094391/ITOF?u=ogde72764&sid=summon&xid=9c803505 (accessed 21 December 2024).

59 Amazon Introduces AI Feature (2022). *Communications Today*. 30 November 2022, NA. Gale Business: Insights https://link.gale.com/apps/doc/A728347182/GBIB?u=ogde72764&sid=summon&xid=642f7046 (accessed 21 December 2024).

60 Kepuska, V. and Bohouta, G. (2018). Next-generation of virtual personal assistants (Microsoft Cortana, Apple Siri, Amazon Alexa and Google home). In: *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, 99–103. IEEE.

61 Morgan, J. (2023). *How AI Will Make Payments More Efficient and Reduce Fraud*. https://www.jpmorgan.com/insights/payments/payments-optimization/ai-payments-efficiency-fraud-reduction.

62 NEC and Siemens Join Forces to Provide AI Monitoring and Analysis Solution to Manufacturers (2020). *Manufacturing Close-Up*, March 12, 2020, NA. Gale Business: Insights. https://go.gale.com/ps/i.do?p=GBIB&u=ogde72764&id=GALE%7CA617136149&v=2.1&it=r&sid=summon (accessed 21 December 2024).

63 Mendonça, E. (2023). *John Deere Revolutionizes Agriculture with AI and Automation*. https://www.assemblymag.com/articles/97831-john-deere-revolutionizes-agriculture-with-ai-and-automation.

64 Tejani, A.S., Retson, T.A., Moy, L. et al. (2023). Detecting common sources of AI bias: questions to ask when procuring an AI solution. *Radiology* 307 (3): e230580.

65 Zhang, Y. and Gosline, R. (2023). Human favoritism, not AI aversion: people's perceptions (and bias) toward generative AI, human experts, and human–GAI collaboration in persuasive content generation. *Judgment and Decision Making* 18: e41.

66 Ungureanu, C.T. and Amironesei, A.E. (2023). Legal issues concerning generative AI technologies. *Eastern Journal of European studies* 14 (2): 45–75.

67 Ferreres, A.R. (2024). Ethical and legal issues regarding artificial intelligence (AI) and management of surgical data. *European Journal of Surgical Oncology* 108279–108279.

68 Generative AI and The Future of Law Firm Management (2023). *Bar & Bench*, Gale General OneFile. https://go.gale.com/ps/i.do?p=ITOF&u=ogde72764&id=GALE%7CA762038651&v=2.1&it=r&sid=summon&asid=1eda0982 (accessed 21 December 2024).

69 Myers, A. (2023). *AI-Detectors Biased Against Non-Native English Writers*. https://hai.stanford.edu/news/ai-detectors-biased-against-non-native-english-writers.

70 Liang, W., Yuksekgonul, M., Mao, Y. et al. (2023). GPT detectors are biased against non-native English writers. *Patterns* 4 (7): 100779.

71 Corless, V. (2023). *AI Detectors Have a Bias Against Non-Native English Speakers*. https://www.advancedsciencenews.com/ai-detectors-have-a-bias-against-non-native-english-speakers/.

72 Najarro, I. (2023). *Another AI Issue for Schools to Know About: Bias Against Non-native English Speakers*. https://www.edweek.org/technology/another-ai-issue-for-schools-to-know-about-bias-against-non-native-english-speakers/2023/08.

73 Zwart, H.D. (2024). *Racist Technology in Action: ChatGPT Detectors are Biased Against Non-native English Writers*. https://racismandtechnology.center/2024/03/04/racist-technology-in-action-chatgpt-detectors-are-biased-against-non-native-english-writers/.

74 Vainilavičius, J. (2023). *AI Detection Tools Biased Against Non-Native English Speakers, Study Shows*. https://cybernews.com/tech/ai-detectors-discriminate-english/.

75 Tabatabaian, M. (2024). Prompt engineering using ChatGPT: crafting effective interactions and building GPT apps. *Mercury Learning and Information*. https://books.google.com/books?id=mtYJEQAAQBAJ.

76 Kochhar, R. (2023). *Which U.S. Workers are More Exposed to AI on Their Jobs*? https://www.pewresearch.org/social-trends/2023/07/26/which-u-s-workers-are-more-exposed-to-ai-on-their-jobs/.

77 Cooban, A. (2024). *AI will Shrink Workforces Within Five Years, Say Company Execs*. https://www.cnn.com/2024/04/05/business/ai-job-losses/index.html.

2

Fundamentals of Deep Learning

2.1 Neural Networks Overview

A neural network is an artificial intelligence algorithm that has interconnected layers of functions designed to learn complex tasks, resembling neuron activities in human brains. It possesses the ability to learn, or in other words, find optimal model weights or biases through a process called training. Although the way our brain learns is highly complex and not properly understood, the way a neural network learns through training is somewhat similar to the learning process observed in humans and other animals. This resemblance is why the term "neural" is used in the fields of machine learning and deep learning. A neural network consists of several key components, such as the number of input, output, and hidden layers, and the types of activation functions. These components form the foundation of the network's architecture and determine its performance in prediction or forecasting. A basic multilayer neural network example is shown in Figure 2.1, which consists of an input layer, three hidden layers with biases, and an output layer.

To train a neural network for practical applications, a dataset with real-world inputs and outputs is essential. Let's consider a dataset that includes input and output data of multiple lathe machines that are used in a company. For example, inputs might include the manufacturing year, power consumption, load, maintenance history, operational hours, operating temperature, and vibration levels. The outputs could be the remaining life of the machine, the probability of failure, and the maintenance schedule. After successful training, the machine learning algorithm should be capable of predicting the outputs for unknown data of a similar type of lathe machine. Through the process of training the neural network, the model can establish the underlying patterns and connections that exist between the input variables and the corresponding output variables. After the training phase, if the model has undergone proper validation, it should possess the capability to generate accurate predictions of output values when presented with input data similar to the training dataset. However, it is important to ensure that the dataset used for training is extensive in size and encompasses a wide range of variations to make reliable predictions.

2.2 Basic Concepts and Terminologies

2.2.1 Neurons and Layers

The neuron is the basic computational node of a neural network that applies mathematical functions to process the inputs and produce output. Each neuron can receive one or more inputs, which can be the initial inputs to the network or the outputs from the neurons of the previous layer.

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

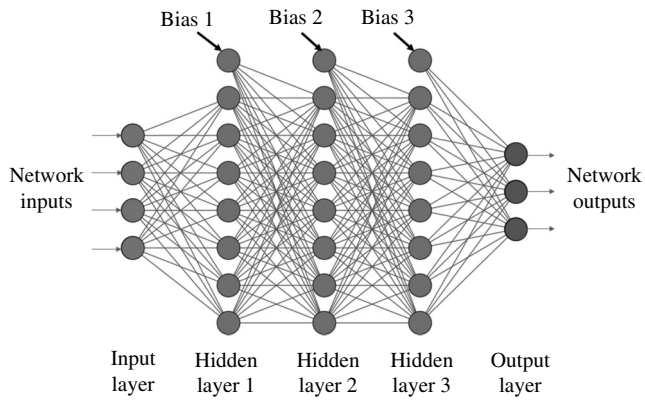


Figure 2.1 Multi-layer neural network architecture with three hidden layers. *Source:* Ref. [1]/Taylor & Francis Group.

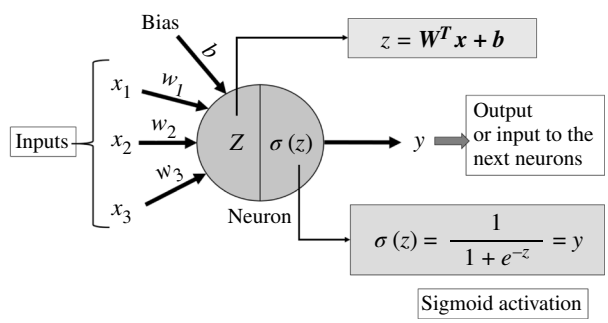


Figure 2.2 Computational steps inside one neuron of the hidden layer. Here, $\sigma(z)$ is the sigmoid activation; x_1, x_2, x_3 are inputs to the neuron; w_1, w_2, w_3 are the weights, and b is the bias to the input [2].

Neurons are organized into layers, with each neuron in a layer connected to the neurons in the following layers. Inside the neuron, a suitable activation function is used, which determines the output and introduces nonlinearity to the model. Figure 2.2 shows the operations inside one of the neurons in hidden layers. This operation can be broken down into two computational steps. First, z is determined using the weights (W), bias (b), and inputs (x). Second, the activation function $\sigma(z)$ is computed using z. In the example shown in Figure 2.2, a sigmoid activation function is used to demonstrate how a neuron works. Here, we could describe the output of a neuron as a function of a function.

To illustrate how a neuron works mathematically, let's consider three input features, $x_1 = 0.1$, $x_2 = -0.2$, and $x_3 = 0.3$, and the associated weights for these inputs, $w_1 = 0.5$, $w_2 = -0.6$, and $w_3 = 0.7$. If a bias term $b = 0.1$ is used, the neuron will calculate the weighted sum of the inputs using Eq. (2.1).

$$z = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b} \tag{2.1}$$

The neuron will then apply an activation function (e.g. sigmoid function) to this weighted sum to calculate the output from the neuron. This calculation is illustrated in Figure 2.3. It demonstrates the use of Eq. (2.1) to compute the z value, which is then passed through the sigmoid function, resulting in an output of 0.6177. The sigmoid function is presented in Eq. (2.2).

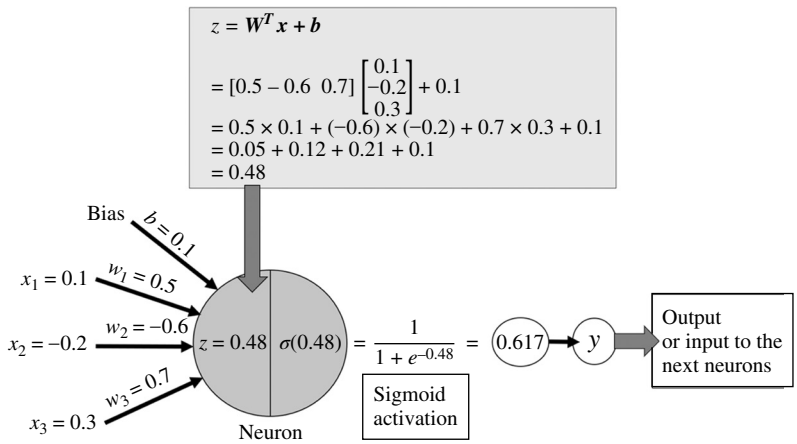


Figure 2.3 Example calculation inside a single neuron, where $x_1 = 0.1$, $x_2 = -0.2$, and $x_3 = 0.3$ are inputs, and associated weights for these inputs are $w_1 = 0.5$, $w_2 = -0.6$, and $w_3 = 0.7$, and the bias term $b = 0.1$. Here, the z value is processed through a sigmoid function to get the output (0.6177) from the neuron.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

In a neural network, this process is repeated in neurons across hidden layers to model complex and nonlinear relationships between inputs and outputs. If a neural network contains many hidden layers, it is referred to as deep learning. Therefore, all deep learning networks are a type of neural network. Deep learning typically demands greater computational resources due to its complex and lengthy networks with ten to hundreds of hidden layers and many nodes in each layer. By linking multiple neurons in layers and fine-tuning the weights and biases during training, deep learning can perform tasks such as classification, detection, segmentation, and image and text generation, etc.

The sigmoid activation function can transform any predicted value into a range between 0 and 1. This output helps estimate probabilities with a threshold (such as 0.5) in binary classification tasks. For example, if the output is $\geq$0.5, the class is assigned as 1; if the output is <0.5, the class is set to 0. Various other activation functions can be applied in artificial neural networks. Choosing the right activation function is crucial, as it significantly affects the training performance.

2.2.2 Role of Activation Functions

The activation function introduces nonlinearity to the model so that it can learn complex patterns from the dataset. During the training process, it transforms the weighted sum of inputs at each neuron into an output signal and determines whether the neuron should be activated and to what extent. Generally, activation functions are located in the hidden and output layers, and various activation functions are available based on training types. For example, the sigmoid and softmax functions are typically used for binary and multi-class classification problems, as they can map output values between 0 and 1. The ReLU (Rectified Linear Unit) activation and its variants, such as Leaky ReLU, are used due to their ability to mitigate vanishing gradient problems and apply sparse activation. The hyperbolic tangent (tanh) function is used when normalized outputs are needed for model training.

The choice of activation function plays a significant role in determining the efficiency and accuracy of the model training. Several comprehensive lists of activation functions for PyTorch, TensorFlow, and Keras libraries are available at [3–5].

2.2.3 Learning Types

Deep learning networks utilize various learning methods depending on data types. Some common examples are supervised, unsupervised, semi-supervised, and reinforcement learning. Supervised learning occurs when the network can adjust its parameters by comparing its predictions to the correct outputs provided in the dataset. For example, in computer vision, when a network is trained on a dataset of different types of fruits and learns to predict fruit types from the input image or video feed, it is called supervised learning. Other examples of supervised learning include medical diagnosis using patient records, speech recognition systems trained on labeled voice data, and email spam detection based on pre-classified emails.

On the other hand, unsupervised learning is the type of learning when the required outputs or correct results from the training dataset are not labeled [6]. In unsupervised learning, the network autonomously identifies key patterns or features in the data. Examples of unsupervised learning include image segmentation without pre-segmented training data, anomaly detection in machinery without prior failure data, etc. Semi-supervised learning falls between the supervised and unsupervised methods, and it uses datasets that are only partially labeled [7].

Reinforcement learning is another type of learning method that makes decisions based on rewards or penalties received from a complex environment [8]. It utilizes observations from a complex environment and makes a final decision (output) based on the penalties or rewards it received for its performance. In deep learning, the choice of various learning schemes depends on the nature of the available data and the specific requirements of a given task.

2.3 Training on a Network

2.3.1 Forward and Backward Propagation

Training a deep learning model on a network involves an iterative process of forward propagation and backpropagation, which enables the model to learn from input–output parameters. During forward propagation, input data travels through multiple layers of neurons in the network, where each neuron's output is calculated by applying a weighted sum of the inputs, adding a bias, and passing the result through an activation function. The network's predictions are then compared to target outputs using a loss function.

In the backpropagation, gradients of the loss function with respect to the network's weights and biases are computed by propagating errors backward. This process employs the chain rule of differentiation to calculate partial derivatives across layers. The gradients enable weight and bias values to be updated using optimization algorithms such as gradient descent or Adam. The goal of backpropagation is to minimize the loss function. For example, in image classification, forward propagation might process pixel values through convolutional and fully connected layers to predict object categories. If the model misclassifies an apple as an orange, backpropagation would adjust weights to strengthen features indicative of "apple" and weaken those leading to the "orange" classification. Similarly, in natural language processing, a sequence-to-sequence model translating English to French would use forward propagation to generate a translation, and then translation errors from backpropagation are utilized to refine its understanding of linguistic patterns and vocabulary use. In deep learning, this cyclical process of prediction, error calculation, and parameter adjustment continues until the model achieves a satisfactory level of performance on the training data.

2.3.2 Loss Functions

The purpose of the loss function is to quantify how well a model's predictions align with the true values in the training data. It serves as the primary feedback mechanism for optimizing the model through the training process. The loss function calculates errors and enables the network to adjust its parameters (weights and biases) to achieve better performance. For this reason, it is also called the objective or error function. For example, in a binary classification task, such as spam email detection, the binary cross-entropy loss can be used. As the model processes email data from training datasets, it outputs a probability of each email being spam. The loss function compares these probabilities to the actual labels (spam or not spam) and gives a score to the model that reflects its accuracy. In a more complex scenario, such as image classification with multiple classes, cross-entropy loss can be used. If the model is trained for classifying objects in images (e.g. cars, trucks, buildings), the loss function would evaluate how close the model's probability distribution over all possible classes is compared to the real classes available for each image.

The loss function is often referred to as the cost function when the error is computed for the entire training set or a batch. A standard form of the binary cross-entropy loss function is represented by Eq. (2.3) [9].

$$J = -\frac{1}{M}\sum_{m=1}^{M}[y_m \times \log(h_\theta(x_m)) + (1 - y_m) \times \log(1 - h_\theta(x_m))] \tag{2.3}$$

where M = number of training examples, y_m = target label for training example m, x_m = input for training example m, h_θ = model with neural network weights θ. For categorical targets, if $\hat{y}_1 \ldots \hat{y}_m$ are the probabilities of the "m" classes, and the rth class is the ground-truth class, then the loss function for a single instance is defined by Eq. (2.4) [10].

$$J = -\log(\hat{y}_r) \tag{2.4}$$

The objective of any network is to minimize the expression given in Eq. (2.4). This minimization is achieved by adjusting the weights through an optimization function or optimizer. Therefore, the choice of an appropriate loss function is very critical, and it depends on the specific computational context for different datasets.

The cross-entropy loss is widely used in classification tasks. It is very effective when the training goal is to predict probabilities, as it penalizes models heavily for low-confidence predictions of the correct class. Besides cross-entropy loss, some other common loss functions are mean squared error (MSE), mean absolute error (MAE), negative log-likelihood loss, Huber loss (combines MSE and MAE), hinge loss (used in support vector machines), etc.

2.4 Gradient Descent Algorithm

Gradient descent is an iterative optimization technique used in neural networks to minimize the cost function [11]. Unlike linear regression, where derivatives can locate the minimum of convex functions, neural networks need to deal with non-convex functions. This complexity means that gradient descent may converge on local minima rather than the global minimum. Although gradient descent is a powerful tool for optimizing neural networks, it doesn't guarantee finding the absolute best solution (global minimum) due to the intricate, non-convex nature of deep learning. It might settle on a good (but not optimal) solution (local minimum). For example, let's assume a

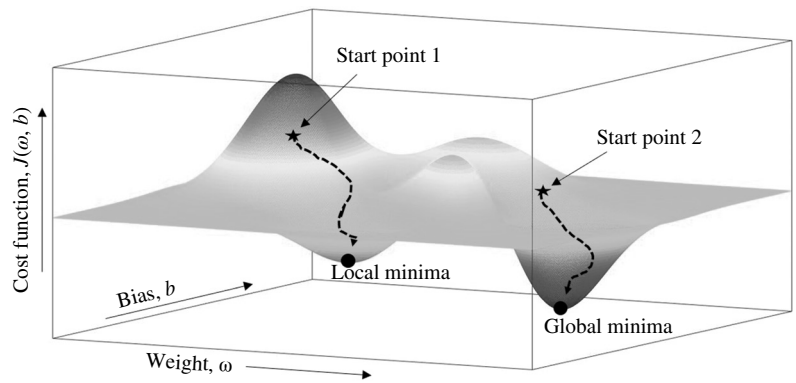


Figure 2.4 If the cost function, *J*, is a function of two parameters (weight and bias), the gradient descent algorithm will search for a minima on the surface. In the example plot, starting from Point 1 leads to convergence at a local minimum, and starting from Point 2 results in reaching the global minimum. *Source:* Ref. [2]/Springer Nature.

neural network is trained to recognize handwritten digits, and gradient descent helps to improve its performance by adjusting the network's parameters. Here, if the network can't find the optimum weights through training, it might find a solution that's very good at recognizing most digits but struggles with a few specific variations of handwriting.

Figure 2.4 shows how the gradient descent can lead to different minima (solutions) depending on the initial starting point. In this 3D surface plot, the cost function, *J*, depends on two parameters (ω and b) and forms a complex shape. Here, the height of the surface represents the value of the cost function *J* for each corresponding weight and bias.

In many machine learning optimization problems, the cost function exists in a high-dimensional space because of the numerous weights and biases involved. The gradient descent algorithm can update all the weights and biases simultaneously. For example, if we have n neurons in a hidden layer, all the weights will update through Eq. (2.5) [12]. In this equation, α is the learning rate, which tells the algorithm how big or small the step size would be while moving in the direction of downhill steep. The negative gradients push the algorithm to move toward the downhill direction.

$$\begin{aligned}
\omega_1 &= \omega_1 - \alpha \frac{\partial}{\partial \omega_1} J(\omega, b) \\
\omega_2 &= \omega_2 - \alpha \frac{\partial}{\partial \omega_2} J(\omega, b) \\
\omega_3 &= \omega_3 - \alpha \frac{\partial}{\partial \omega_3} J(\omega, b) \\
\omega_n &= \omega_n - \alpha \frac{\partial}{\partial \omega_n} J(\omega, b)
\end{aligned} \tag{2.5}$$

The bias parameters for a hidden layer can be updated using Eq. (2.6).

$$b = b - \alpha \frac{\partial}{\partial b} J(\omega, b) \tag{2.6}$$

Equations (2.5) and (2.6) illustrate the calculation step for a single layer, but this process becomes significantly more complex and computationally intensive in deep learning when multiple hidden layers are involved. In this case, the learning rate α is an important hyperparameter that requires careful tuning. A learning rate that is too small can make the gradient descent algorithm inefficient

and may prevent it from reaching a global solution. On the other hand, a large learning rate could cause the algorithm to overshoot the minima, preventing proper convergence.

Gradient descent algorithms can have many different forms based on how they update or adjust learning rates. For example, batch gradient descent computes the gradient of the cost function using the entire dataset. Although it can provide stable updates, it becomes computationally intensive for large datasets. Stochastic Gradient Descent updates the network parameters after processing each training example. This approach can lead to a faster but noisier update. The mini-batch gradient descent combines the two approaches by updating the parameters after processing a small batch of data and balancing between efficiency and stability. Adaptive Gradient Algorithm (AdaGrad) adapts the learning rate for each parameter based on previous gradients. This approach is effective for sparse data, such as in natural language processing models. Root Mean Square Propagation (RMSprop) scales the learning rate for each parameter based on the magnitude of recent gradients and is well-suited for recurrent neural networks in tasks like time series prediction. Adam, an improvement on AdaGrad and RMSprop, uses both the mean and variance of the gradients to adapt learning rates. Therefore, Adam is very suitable for training convolutional neural networks (CNNs).

In addition to gradient descent, various other optimization techniques are available for machine and deep learning. These include evolutionary approaches such as genetic algorithms, which mimic natural selection to evolve neural network architectures [13]. Probabilistic methods such as simulated annealing explore the solution space by allowing occasional uphill moves to escape local optima [14]. Swarm intelligence techniques such as Particle Swarm Optimization (PSO) use collective behavior to search for optimal solutions [15]. The Levenberg–Marquardt algorithm combines Gauss–Newton and gradient descent methods for faster convergence in some scenarios [16]. Bayesian Optimization leverages probabilistic models to efficiently search high-dimensional spaces [17].

The field of deep learning optimization is continuously evolving, and researchers are constantly developing and refining algorithms. The choice of a good optimization depends on many factors. It should be selected by considering computational resources, dataset characteristics, and the specific requirements of the task at hand.

2.5 Weight Initialization and Regularization

Weight initialization and regularization are crucial techniques in deep learning that significantly impact model performance and generalization. Weight initialization sets the starting point for a neural network's learning process. For example, in an image classification task, if all weights start at zero, every neuron would compute the same output, making the network unable to learn diverse features. Similarly, if all neurons start with the same weights, the network's weight updates would be symmetric, which will prevent the network from learning distinct patterns in the data. To avoid these issues, we can choose small random values at the start. This approach will allow neurons to start with slightly different computations and will enable them to learn various aspects of the input data. There is another method called Xavier or Glorot initialization. This method scales the random values based on the number of input and output connections, which helps maintain consistent variance across layers [18]. It is generally used in networks with sigmoid or tanh activations. Besides randomness, it is also important to avoid large weights at the beginning. This can lead to the exploding gradient problem, where gradients become excessively large during backpropagation, pushing weight updates to extreme values. On the other hand, very small weights can cause

the vanishing gradient problem, where the gradients shrink significantly, slowing or halting learning. Therefore, a proper weight initialization approach is essential for learning meaningful features during model training.

Recently, several methods have been developed to address various initialization issues, such as LeCun initialization, truncated normal distribution, Xavier Glorot initialization, and Kaiming He initialization [19–22]. These approaches aim to balance weight distribution and prevent gradients from becoming too large or too small. Popular frameworks such as TensorFlow, PyTorch, and Keras offer built-in functions to handle weight initialization effectively, ensuring that the values fall within a suitable range.

Weight regularization is a technique designed to combat overfitting during training by adding constraints to the model's weights. Overfitting occurs when the model becomes too complex, which leads to overly large weights that fit the training data too closely but fail to generalize to unseen data. Regularization helps by adding a penalty term to the loss function that discourages the model from producing excessively large weights. The regularization parameter is usually denoted by λ, which is a small number, and a cost function can be obtained after regularization, as shown in Eq. (2.7).

$$Cost = Loss\ Function + \frac{\lambda}{2m}\sum_{i=1}^{n}\left\|w^{[i]}\right\|^2 \tag{2.7}$$

where λ = regularization parameter, m = number of inputs, n = number of layers, $w^{[i]}$= weight matrix for ith layer. By minimizing the augmented error with a small regularization term λ, excessively large or complex weights are reduced. The regularization method described in Eq. (2.7) is referred to as L2 regularization because it penalizes the sum of the squared weights. L2 regularization is commonly applied in most scenarios, but for feature selection tasks, L1 regularization is sometimes preferred. In L1 regularization, as shown in Eq. (2.8), the sum of the absolute values of the weights is penalized instead.

$$Cost = Loss\ Function + \frac{\lambda}{2m}\sum_{i=1}^{n}\left\|w^{[i]}\right\| \tag{2.8}$$

Both weight initialization and regularization require careful tuning. They act like adjusting the starting point and the rules of the game for deep neural networks, and can significantly influence how well the model will learn and generalize new data.

2.6 Hyperparameter Tuning

Hyperparameters are parameters in the deep learning network that are not learned during training but significantly impact how effectively the model learns. For example, the algorithm can adjust weights and biases to reduce the loss function during the training process, but it has no control over other pre-set parameters, such as the number of hidden layers, the number of neurons in each layer, or the choice of activation functions. These predefined parameters that shape the model's performance are referred to as hyperparameters.

Hyperparameter tuning in machine learning is a complex process that relies heavily on experimentation and experience. The best hyperparameters are often specific to the problem and dataset being used. Most of the time, strategies that work well in one area, such as computer vision, do not necessarily work well in another, such as natural language processing. Key hyperparameters in deep learning include the learning rate, batch size, number of hidden layers, and regularization

methods. During the training process, some hyperparameters may have minimal impact, and others can dramatically affect the outcome. We typically need to balance factors such as model complexity, computational limitations, and the speed of convergence when selecting values. Overall, successful hyperparameter tuning requires a mix of systematic testing, domain expertise, and practical constraints related to the problem to maximize model performance.

2.6.1 Architecture-based Hyperparameters

Architecture-based hyperparameters are defined by the structure and composition of the model itself, such as the number of layers in a neural network, number of neurons in each layer, type of activation functions, kernel size in convolutional neural networks, etc. These hyperparameters essentially shape the model's capability, define its approach to processing input data, and generate meaningful patterns.

2.6.2 Training-based Hyperparameters

Training-based hyperparameters control how the model learns during the training process. Examples of training-based hyperparameters are learning rate, batch size, number of epochs, optimizer types, regularization techniques, etc. Training-based hyperparameters govern the overall learning dynamics of the model. They influence the speed at which the model reaches its optimal state, its ability to strike a balance between memorizing training data and generalizing to new data, and how effectively it utilizes available computing power.

2.7 Overview of Common Hyperparameters

2.7.1 Kernel Size

Kernel size in a CNN significantly impacts the learning process by determining the receptive field (the area of the input image that influences output) of the network. It helps to extract underlying features, such as edges, textures, and shapes, during the process. Smaller kernel sizes, such as 3×3 or 5×5, are typically used to capture fine details (edges or small textures) in the data. A larger kernel size, such as 7 by 7 or 11 by 11, allows the network to capture broader patterns and contexts. For example, in object detection or face recognition tasks, a larger kernel might be better suited for identifying overall general features. On the other hand, smaller kernel sizes can be suitable for tasks requiring attention to minute textures or edges, such as detecting small defects in manufacturing quality control. Therefore, kernel size influences how the network balances its focus between small-scale details and broader patterns.

2.7.2 Learning Rate

The learning rate is an important hyperparameter in deep learning. It controls how quickly (or cautiously) a model updates its parameters based on the gradients. A high learning rate (e.g. 0.1) can accelerate training by implementing larger updates in the direction of the optimal solution. For example, in an image classification task, this might lead to rapid improvements in accuracy during early epochs. However, it also risks overshooting the optimal point as the model solution will oscillate close to the best solution. On the other hand, a low learning rate (e.g. 0.0001) promotes more stable and

precise updates. However, it converges using smaller steps, which may lead to very slow training, and the solution may get stuck in local minima. In natural language processing tasks, a low learning rate can help the model learn subtle semantic differences. But it also leads to very slow convergence, and there is a risk of getting stuck in a suboptimal solution. There are some adaptive learning rate techniques, such as Adam or RMSprop, that may help to strike a balance by adjusting the learning rate. The deep reinforcement learning agent for game playing typically uses adaptive methods to implement larger updates when the agent is far from optimal behavior and smaller updates for more refined approach as the model starts learning. Therefore, finding a good learning rate is critical, and we need to pay careful attention to this hyperparameter during model training.

2.7.3 Weight Decay

Weight decay is a very effective regularization method that employs a penalty coefficient (often represented by λ) to each layer to reduce the weights and overfitting. This approach is essentially a variant of L2 regularization, as it originated from the gradient of the L2 norm of weights in gradient descent. In certain cases, using a single weight decay coefficient is more efficient because it reduces the number of hyperparameters to be tuned in the training.

2.7.4 Dropout

Dropout is an effective regularization technique used to address the overfitting problem. It randomly and temporarily deactivates a predefined percentage of neurons in each layer at every iteration. This scaled-down process creates a smaller, simplified version of the original neural network. This kind of random deactivation has been shown to be very effective in mitigating overfitting, as it encourages the network to develop a more generalized set of weights. Dropout has been shown to reduce overfitting and produce a more diverse distribution of weights, ultimately leading to better generalization in neural networks.

We can see the impact of dropout in tasks such as image classification, where it prevents the model from becoming overly reliant on specific pixels or features. Therefore, it is important to consider dropout techniques if model training starts to overfit the training data.

2.7.5 Batch, Iteration, and Epochs

Batch, iteration, and epoch are key hyperparameters in deep learning that significantly impact model training and performance. Their selection should account for the nature of the dataset and the available computational resources.

A batch is a portion of the training data that is fed into the model at once instead of using the full dataset. Using batches of data helps to avoid computationally intensive operations. In deep learning, using batches is essential, as feeding the entire dataset at once is usually impractical due to the memory constraints of GPUs.

Let's consider a scenario where we're training a natural language processing model to classify customer reviews as positive or negative using a dataset of 500000 reviews. A batch represents a subset of the training data processed at once. For example, with a batch size of 5000 review data, the model processes 5000 reviews simultaneously, updating its understanding of sentiment patterns. This approach is more memory-efficient and allows for more frequent model updates compared to processing all 500000 reviews at once.

An iteration occurs each time the model updates its parameters based on a batch. In our example, if we use a batch size of 5000, it would take 100 iterations to process all 500000 reviews at once. During each iteration, the model refines its ability to comprehend positive from negative language in the reviews.

An epoch represents a complete cycle through the entire dataset. With our 500000 reviews and a batch size of 5000, one epoch would consist of 100 iterations. If we train for 10 epochs, the model will see each review 10 times, and its sentiment classification accuracy will gradually improve. Using too many epochs in training might lead to overfitting, where the model becomes too accurate in classifying the training reviews and performs poorly on new, unseen reviews.

Balancing these hyperparameters is crucial. For example, smaller batch sizes might allow for more nuanced learning but require more iterations, and larger batch sizes might speed up training but may miss subtle patterns in the reviews. Therefore, the number of epochs needs to be high enough for learning and generalization but not so high that it starts to memorize the training data.

2.8 Challenges and Best Practices for Efficient Tuning

Efficient deep learning training and fine-tuning present several challenges, and we will need to adhere to best practices at the beginning. Some of the key challenges include managing overfitting, handling large datasets, and balancing computational resources. Also, the vast hyperparameter space can lead to time-consuming and computationally expensive search processes. Therefore, it is crucial to optimize resource usage, especially on GPUs, based on computational powers.

To address these challenges, practitioners often use regularization techniques, such as dropout or weight decay, to prevent overfitting, utilize adaptive optimizers, such as Adam, for stable learning, and often employ techniques, such as cross-validation and early stopping, to avoid unnecessary cycles. Sometimes, automated hyperparameter optimization techniques, including Bayesian optimization, random search, and grid search, can help navigate the parameter space more efficiently than manual fine-tuning. We also need to utilize transfer learning as much as possible. In those cases, we can start from pre-trained model weights that are already fine-tuned for specific tasks. This approach can significantly reduce training time and improve performance, especially with limited data.

To speed up the training process, we should consider implementing efficient data loading and preprocessing, along with distributed training through multiple GPUs or machines. Also, continuously monitoring and printing validation metrics throughout the training process helps to ensure that the model generalizes well. Lastly, maintaining well-documented training experiments and using version control for both code and model checkpoints enables reproducibility and facilitates model performance. By combining these practices and exploring updates on the latest optimization techniques, deep learning practitioners can effectively navigate the challenges of model tuning and achieve improved training performance.

References

1 Arif, T. M., & Rahim, M. A. (2024). *Deep Learning for Engineers*. Boca Raton: CRC Press.
2 Arif, T.M. (ed.) (2020). Basic artificial neural network and architectures. In: *Introduction to Deep Learning for Engineers: Using Python and Google Cloud Platform*, 17–27. Cham: Springer International Publishing.

3 PyTorch (2024). torch.nn. https://pytorch.org/docs/stable/nn.html.
4 TesnsorFlow (2024). Module: tf.keras.activations. https://www.tensorflow.org/api_docs/python/tf/keras/activationshttps://www.tensorflow.org/api_docs/python/tf/keras/activations.
5 Keras (2024). Layer activation functions. https://keras.io/api/layers/activations/.
6 Sun, R., Hou, X., Li, X. et al. (2022). Transfer learning strategy based on unsupervised learning and ensemble learning for breast cancer molecular subtype prediction using dynamic contrast-enhanced MRI. *Journal of Magnetic Resonance Imaging* 55 (5): 1518.
7 Hagberg, E., Hagerman, D., Johansson, R. et al. (2022). Semi-supervised learning with natural language processing for right ventricle classification in echocardiography—a scalable approach. *Computers in Biology and Medicine* 143: 105282.
8 Pateria, S., Subagdja, B., Tan, A.H. et al. (2021). Hierarchical reinforcement learning: a comprehensive survey. *ACM Computing Surveys* 54 (5): 1–35.
9 Shang, W., Sohn, K., Almeida, D. et al. (2016). *Understanding and Improving Convolutional Neural Networks via Concatenated Rectified Linear Units*. https://arxiv.org/abs/1603.05201.
10 Gupta, A. and Duggal, R. (2017). P-TELU: parametric tan hyperbolic linear unit activation for deep neural networks. In: *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*, 974–978, 22–29 October 2017. https://ieeexplore.ieee.org/document/8265328.
11 Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. Cambridge: MIT Press.
12 Liew, S.S., Khalil-Hani, M., and Bakhteri, R. (2016). Bounded activation functions for enhanced training stability of deep neural networks on visual pattern recognition problems. *Neurocomputing* 216: 718–734.
13 Shirvani, A., Nili-Ahmadabadi, M., and Ha, M.Y. (2024). A deep learning–genetic algorithm approach for aerodynamic inverse design via optimization of pressure distribution. *Computer Methods in Applied Mechanics and Engineering* 429: 117187.
14 Ma, Z., He, X., Yan, P. et al. (2023). A fast and flexible algorithm for microstructure reconstruction combining simulated annealing and deep learning. *Computers and Geotechnics* 164: 105755.
15 Tan, T.Y., Zhang, L., and Lim, C.P. (2019). Intelligent skin cancer diagnosis using improved particle swarm optimization and deep learning models. *Applied Soft Computing* 84: 105725.
16 Narendra, M., Valarmathi, M.L., Anbarasi, L.J. (2024). Levenberg-Marquardt deep neural watermarking for 3D mesh using nearest centroid salient point learning. *Scientific Reports* 14 (1): 6942–6942.
17 Tang, S., Zhu, Y., and Yuan, S. (2022). Intelligent fault diagnosis of hydraulic piston pump based on deep learning and Bayesian optimization. *ISA Transactions* 129: 555–563.
18 Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the* 13th *International Conference on Artificial Intelligence and Statistics*, PMLR 9: 249–256. https://proceedings.mlr.press/v9/glorot10a.html.
19 Bagheri, R. (2020). *Weight Initialization in Deep Neural Networks*. https://towardsdatascience.com/weight-initialization-in-deep-neural-networks-268a306540c0.
20 Rubin, O. (2023). *Conversations with GPT-4: Weight Initialization with the Truncated Normal Distribution*.https://medium.com/@ohadrubin/conversations-with-gpt-4-weight-initialization-with-the-truncated-normal-distribution-78a9f71bc478.
21 Hlav, E. (2022). *Xavier Glorot Initialization in Neural Networks – Math Proof*. https://towards-datascience.com/xavier-glorot-initialization-in-neural-networks-math-proof-4682bf5c6ec3.
22 He, K., Zhang, X., Ren, S. et al. (2015). Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. *IEEE International Conference on Computer Vision (ICCV 2015)*, 1502. IEEE Computer Society, USA, 1026–1034. https://doi.org/10.1109/ICCV.2015.123.

3

Convolutional and Recurrent Neural Network

3.1 Introduction

Convolutional neural networks (CNNs) and recurrent neural networks (RNNs) are two major neural network architectures used in deep learning models. CNN primarily deals with spatial datasets, such as images and videos, and RNN, on the other hand, is designed to handle sequential data with the help of an additional memory state. In RNN, the order of the data matters, such as time series, text, audio, etc. In this chapter, we will discuss some key aspects of these two widely used deep learning architectures.

3.2 Historical Background

The origin of CNN can be traced back to the 1980s, when "Neocognitron," a multilayered artificial neural network, was proposed by Kunihiko Fukushima [1]. "Neocognitron" used hierarchical feature extraction through a series of layers, with each layer performing convolution and pooling operations. During that time, the concept of backpropagation in neural networks was popularized through the work of David Rumelhart, Geoffrey Hinton, and Ronald Williams [2]. In 1998, these novel concepts gained momentum again when Yann LeCun and his colleagues developed LeNet-5, which successfully recognized handwritten digits [3]. Although the architecture of LeNet-5 and "Neocognitron" are different, they are both inspired by the human visual system and share conceptual similarities. LeNet-5 utilized multiple layers and subsampling layers followed by fully connected layers and demonstrated impressive performance in recognizing handwritten characters. These developments paved the way for the development of modern CNN architectures and computer vision technology.

CNNs were mostly underutilized until 2012, when AlexNet, developed by Krizhevsky, Sutskever, and Hinton, achieved remarkable success in the ImageNet competition, marking the starting point of the modern deep learning revolution [4]. This breakthrough was followed by rapid advancements in this field, leading to deeper and more sophisticated architectures such as VGGNet (2014), GoogLeNet/Inception (2014), and ResNet (2015) [5–7]. Some specialized architectures such as U-Net for biomedical imaging (2015) and YOLO for real-time object detection (2016) also emerged in the subsequent years [8, 9]. Recent innovations include Mask R-CNN for image segmentation (2017), EfficientNet for optimized architectures (2019), and Vision Transformers (2020) for self attention in image processing, which has demonstrated state-of-the-art performance on various image recognition tasks, often outperform traditional CNNs in terms of accuracy [10–12].

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

The emergence of RNN can also be traced back to the 1980s, when John Hopfield introduced the Hopfield Network, which can store memories as attractors in a dynamical system [13]. Although it was not purely an RNN, it inspired many later developments of recurrent structures. In 1986, David Rumelhart, Geoffrey Hinton, and Ronald J. Williams introduced backpropagation, and Rumelhart and Hinton contributed to the development of the Elman Network, which is also known as Simple RNN or SRNN [2]. From the late 1990s, improvements in overall machine learning algorithms were made possible by a significant rise in computational power. After 2010, deep learning algorithms thrived with the advent of high-performance GPU computing, leading to major breakthroughs in RNN architectures. During this period, long short-term memory (LSTM) became widely popular for analyzing time-dependent data [14]. In 2017, transformer-based models were introduced, and this revolutionized cutting-edge methods such as BERT (Bidirectional Encoder Representations from Transformers) by Google, generative pre-trained transformer (GPT) by OpenAI, and embeddings from language models (ELMo) by AllenNLP [15–17]. These models can be trained on vast datasets and fine-tuned for specific tasks, achieving improved performance. More recently, large language models in ChatGPT and Gemini have set new benchmarks by delivering highly human-like responses [18, 19]. Traditionally, RNN models have also been used in natural language processing to manage, analyze, and detect patterns in sequential data. The recent widespread use of RNNs is delivering state-of-the-art results in many other areas such as speech recognition, video analysis, deepfake generation, music creation, grammatical inference, robot trajectory control, etc.

3.3 Convolutional Neural Networks

3.3.1 CNN Framework

A CNN architecture consists of a sequence of specialized layers that work together to extract and learn hierarchical spatial features from input images. This design enables CNNs to progressively identify more complex visual patterns without manual intervention. The core components of a CNN include convolutional layers for feature detection, pooling layers for dimensionality reduction, and fully connected layers for high-level reasoning and classification. The convolutional layer employs a collection of trainable filters (also known as kernels) that scan across the input image. As these filters move, they perform element-wise multiplication and summation operations, creating feature maps. These resulting maps highlight different visual elements within the image, such as contours, textures, and other distinctive patterns.

3.3.2 Convolutional Operations

In convolutional layers, various types of transformation can be applied to the input image data to extract features. The basic or most common operation involves moving a filter across the input image. At each position, the filter performs element-wise multiplication with the corresponding image region and sums these results. This process generates an output matrix or feature map that emphasizes particular patterns in the image, such as edges, textures, or specific color distributions.

An image in the network is read as an array of numbers representing pixel intensities at different points ranging from 0 to 255. For a color image, three channels are used (Red, Green, and Blue), and each channel is represented as a 2D array of pixel values (Figure 3.1). Therefore, if an input color image is of size 720×480, the input to the CNN will be a tensor of size $720 \times 480 \times 3$. Each layer of the 2D image matrices for Red, Green, and Blue colors is then processed by the CNN.

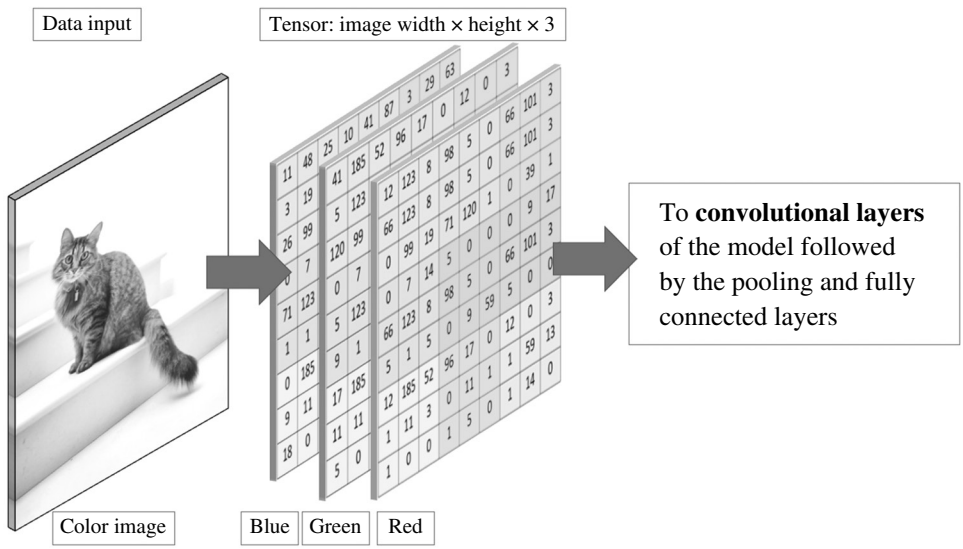


Figure 3.1 A color image is processed by the model using three channels (Red, Green, and Blue) of 2D arrays, where each pixel's color intensity is indicated by a numerical value. These data are then fed into the convolutional layers of the CNN. creativeway/Adobe Stock Photos.

CNNs use a convolution operation where a small filter (or weight) called a kernel (often a 2×2, 3×3, or 5×5 matrix) moves across an input 2D image matrix. At each position, the kernel elements are multiplied with the corresponding image pixels and then summed to produce a single output value. For an input image (I) of size $H \times W$ and a kernel (K) of size $k \times k$, the output at position (i, j) is calculated using Eq. (3.1).

$$\mathrm{O}(i, j) = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} \mathrm{I}(i + u, j + v) * \mathrm{K}(u, v) \tag{3.1}$$

A conceptual illustration of reducing the image matrix size using multiplication and addition is presented in Figure 3.2. If a kernel matrix, K of size 2×2 is used for an image matrix, I of size 3×3, the convolution process will generate a 2×2 output matrix O as it slides over the 3×3 input image. Figure 3.3 shows an illustration of this operation where the kernel filter is applied to each possible position in the input matrix. Using Eq. (3.1), at each position, the kernel is multiplied element-wise with the corresponding submatrix of the input, and the results are summed to generate a single value for the output matrix at that specific position.

The kernel filter in CNN plays a crucial role in performing various image processing operations. It can be designed to perform blurring and sharpening operations and detect edges. These operations are critical to identify key features of an input image.

There are various other types of convolutional operations used in CNN to handle different data types and achieve specific results. These include depth and point-wise convolution for reducing number of parameters and increasing efficiency, dilated convolution for covering a larger receptive field without increasing the number of parameters, grouped convolution for learning more diverse feature representations, 1D convolution for time series or audio data, 3D convolution for learning from volumetric data like videos or 3D medical images, etc. Each convolution operation has its strengths and is selected based on particular needs. These variations in convolution operations also allow CNN users to tailor their networks for optimal performance across a wide range of applications and data structures.

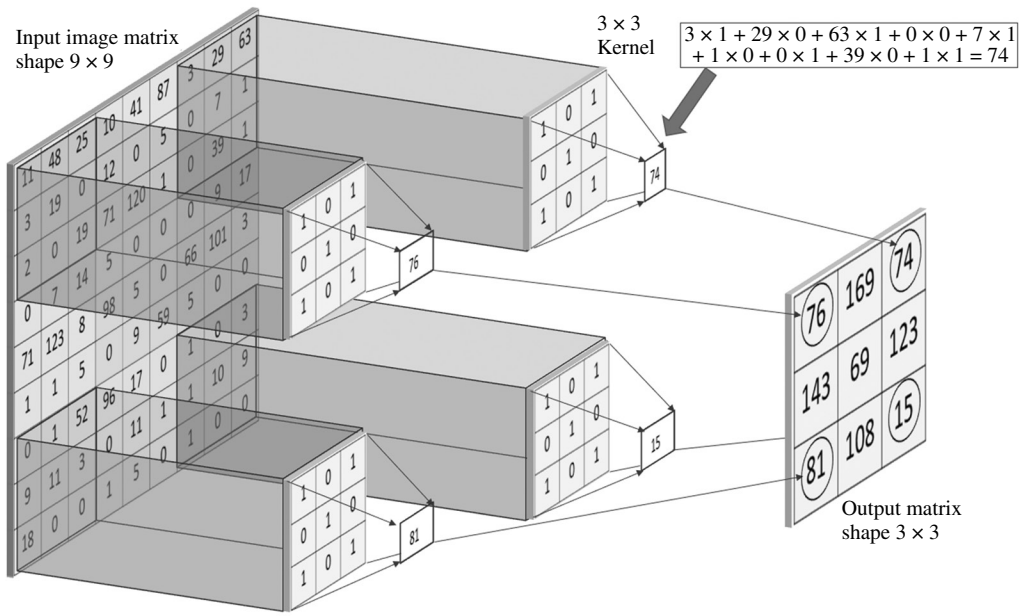


Figure 3.2 An illustration of mapping a 9 × 9 input image matrix using a 3 × 3 convolutional kernel filter.

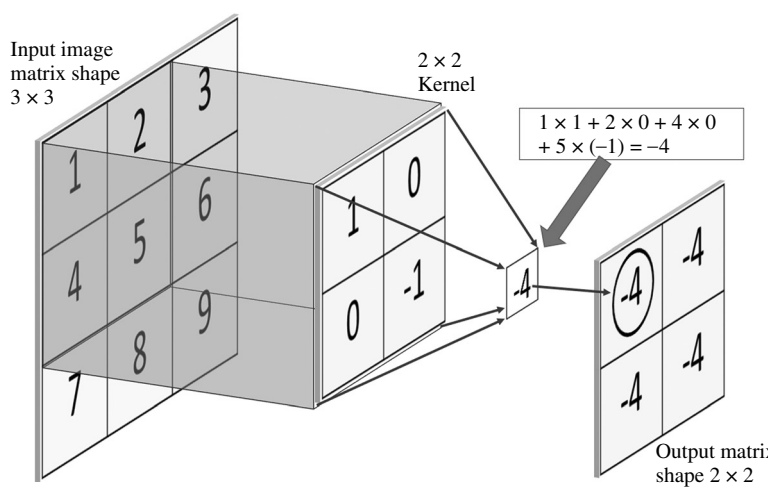


Figure 3.3 An illustration of mapping a 3 × 3 input image matrix using a 2 × 2 convolutional kernel filter.

3.3.3 Padding and Stride

The size of the output matrix depends on the type of padding and stride operation used. Padding is the technique of adding extra pixels, typically zeros, around the input image edges. This practice can control the spatial dimensions of the resulting feature map. Two primary padding types used are valid and same padding. Valid padding (also known as no padding) doesn't add extra pixels and results in a smaller output feature map, as shown in Figures 3.2 and 3.3. Same padding (also known as zero padding) adds enough pixels (usually zeros) around the input to maintain the original spatial dimensions in the output feature map. For example, if a 3 × 3 kernel is used for filtering a

5×5 input image with valid padding, the output matrix will be 3×3. However, if it is used with the same or zero padding, the output matrix will remain 5×5. The choice of padding type affects how the convolution operation interacts with the image edges and can influence the network's ability to detect features near the image boundaries.

Stride is the step size of the kernel as it moves across the input image during convolution. This parameter influences the overlap between the kernel's receptive fields and determines the output matrix dimensions. A stride of one moves the kernel one pixel at a time, resulting in significant overlap and a larger output matrix. On the other hand, a stride of two or more moves the kernel by two or multiple pixels, reducing overlap and producing a smaller output matrix. For example, applying a 3×3 kernel to a 5×5 input image with a stride of one produces a 3×3 output matrix, while a stride of two produces a 2×2 output matrix. Thus, stride serves as a key factor in controlling the spatial reduction and feature extraction process in convolutional layers.

Padding and stride together determine the spatial dimensions of the output feature map and influence the amount of information preserved from the input image. The output dimension using padding and stride can be calculated using Eq. (3.2).

$$\text{Output Dimension} = \left(\frac{I + 2p - k}{s}\right) + 1 \tag{3.2}$$

Here, I is the size of the input image, p is the number of pixels added in the padding, k is the size of the kernel, and s is the stride. For example, for a 5×5 input image with a 3×3 kernel, padding of one, and stride of one, padding will add a border of one pixel around the input image, making it 7×7. The stride will move the kernel one pixel at a time. Using Eq. (3.2), the output matrix dimension will be 5×5, as calculated below.

$$\text{Output Dimension} = \left(\frac{5 + 2 \times 1 - 3}{1}\right) + 1 = 5$$

Also, in Figure 3.2, we have utilized a 9×9 input image with a 3×3 kernel, padding of zero, and stride of three (the kernel moves 3 pixels at a time). Therefore, the Output Dimension is a 3×3 matrix $\left(\left(\frac{9 + 2 \times 0 - 3}{3}\right) + 1 = 3\right)$.

Padding and stride operation offers multiple advantages in CNN. Padding can preserve input image dimension and edge information, which allows deep networks to maintain larger feature maps across layers. It also provides control over the output matrix size. Stride adjusts the convolution step size, which can reduce the dimension of the feature maps, leading to lower computational costs and memory usage. Together, padding and stride offer flexibility in CNN architecture, allowing efficient multi-scale feature extraction and enhancing the network's capacity to learn complex patterns. Using these tools, CNN designers can fine-tune the balance between preserving important features, managing computational resources, and capturing complex spatial relationships.

3.3.4 Pooling Layers

Pooling layer reduces the spatial dimensions of the feature maps produced by convolutional layers. The main purpose of using the pooling layer is to retain the most important information while decreasing the amount of data processed in subsequent layers. This approach is useful in deep learning training to recognize patterns regardless of their position in the image.

Pooling layer operation can be performed in various ways, such as max pooling, average pooling, sum pooling, and L2 pooling. Max pooling selects the largest value from a region, highlighting the most significant feature. Average pooling computes the average value, smoothing the feature map.

Sum pooling adds up the values within a region, and preserve detailed information. L2 pooling takes the square root of the sum of squared values. By reducing the spatial size of feature maps, pooling layers decrease computational complexity. It also reduces the risk of overfitting and makes the network more efficient.

3.3.5 Fully Connected Layers

The final part of a CNN consists of fully connected layers, where the features extracted by earlier convolutional and pooling layers are used for classification or other tasks. These layers do the "decision-making" task of the network. The structure of fully connected layers is similar to that of traditional neural networks. Inside these layers, every neuron is linked to all neurons from the preceding layer. For example, if the previous layer has 50 neurons and the fully connected layer has 10 neurons, there would be 500 (50×10) connections between these two layers. To optimize computational memory, two smaller, fully connected layers are advantageous instead of using one large layer. Well-known networks such as AlexNet and VGG Net utilize multiple fully connected layers, followed by an output layer.

After the fully connected layer, the output is passed through an activation function to produce the final output. The structure of fully connected layers varies depending on the training task, such as classification or segmentation. In some cases, for classification or regression tasks, an alternative supervised method like support vector machines (SVM) can replace fully connected layers.

3.3.6 Output Layers

The CNN's final layer is called the output layer, which employs a specialized function (e.g. softmax for classifying multiple categories) to generate probabilities for each potential class. In this layer, every neuron produces a value ranging from 0 to 1, representing the likelihood of belonging to a particular category.

3.3.7 Overall Architecture

To understand the overall CNN architecture, let's assume that we are training a deep learning model that distinguishes between various car models, and the available training images are labeled with their corresponding make and model. Here, we can use the CNN architecture, where its layers will extract various information at different stages. For example, the first few layers might detect edges and simple shapes, the middle layers would recognize more complex patterns like wheel designs or windshield shapes, and the final layers might identify brand-specific markers such as a Mercedes-Benz star or BMW's kidney grille. After training on many images, the deep learning model will learn to distinguish between various car models. For a multi-class example like this one, we can employ the softmax function or SVM in the final layer. This setup, as shown in Figure 3.4, demonstrates a general layout of a CNN for multi-class classification problems.

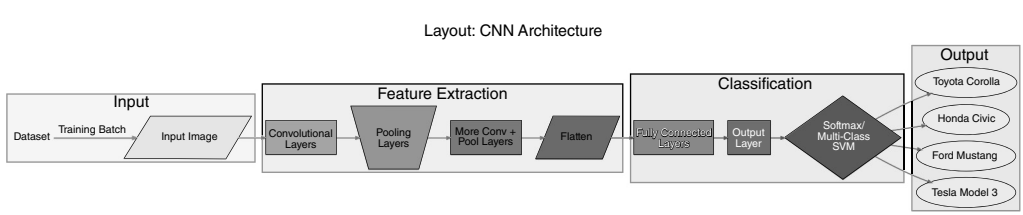


Figure 3.4 A layout of CNN architecture for a multi-class car model classification problem.

3.4 Recurrent Neural Networks

3.4.1 RNN Framework

The RNN is a widely used deep learning structure designed for sequential data prediction. By incorporating an extra memory component into the network, RNNs have demonstrated remarkable effectiveness in speech recognition, sentiment analysis, language translation, music generation, and many more. Within the RNN's internal framework, data flows through hidden layers from one step to the next. As a result, the network's predictions are influenced by present and historical inputs, as well as by previously generated outputs. In contrast to CNNs, RNNs possess the ability to recognize and process relationships between elements across different time points.

3.4.2 Basic Architectures

RNNs are useful in handling vectorized sequential data, and they can adapt to various input–output configurations. Among the common RNN architectures are one-to-many, many-to-one, and many-to-many structures, each designed to process data differently based on the task at hand. For example, the one-to-many architecture transforms a single input into multiple outputs through a series of hidden states, where each state is influenced by its predecessors (Figure 3.5). This structure finds practical application in tasks like image captioning, where a single image input yields a sequence of descriptive words as output.

The hidden layers in RNNs also have distinct characteristics that set them apart from those in CNNs. The primary focus of CNN's hidden layers is on spatial feature extraction using convolutional filters. RNNs, on the other hand, maintain an internal state or memory, which allows them to process sequential data by considering information from previous steps. The temporal processing capability of RNN hidden layers makes them particularly well-suited for tasks involving time series, natural language, or any data with inherent sequential structure.

In the many-to-one RNN architecture, multiple input sequences are processed through hidden states, but only a single output, such as a class or quantity, is produced. This architecture is effective for tasks such as assessing or classifying a person's emotional state or calculating sensitivity scores from written text. A diagram illustrating the many-to-one RNN architecture is provided in Figure 3.6.

A many-to-many RNN architecture has the ability to process multiple input elements and generate multiple output elements through a series of hidden states. This structure finds applications in various domains that require sequence-to-sequence processing. For example, language translation,

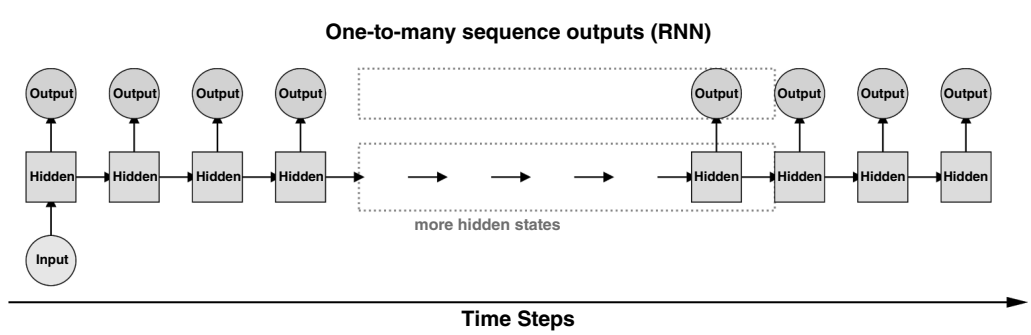


Figure 3.5 Schematic diagram of a one-to-many RNN architecture.

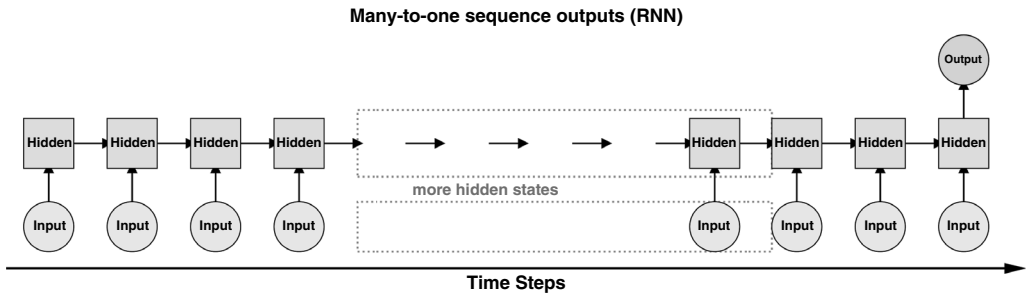


Figure 3.6 Schematic diagram of a many-to-one RNN architecture.

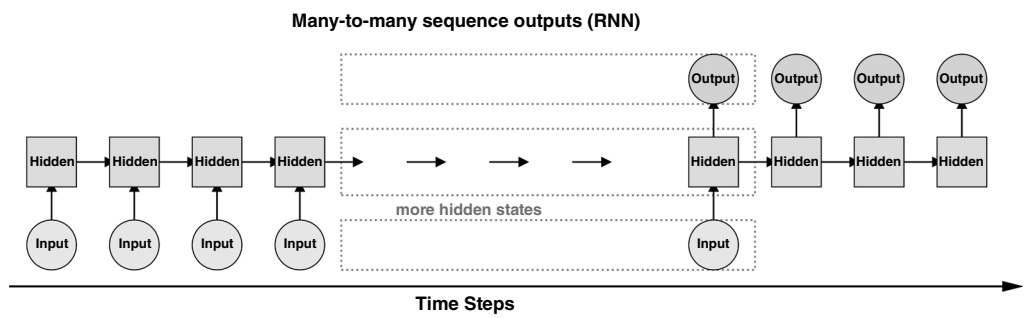


Figure 3.7 Schematic diagram of a many-to-many RNN architecture.

where a sentence in one language is converted into another, video analysis to label individual frames, and various other prediction tasks that involve mapping multiple inputs to multiple outputs.

The configuration of many-to-many RNNs can be flexible, with different arrangements of inputs and outputs to suit specific tasks. Figure 3.7 provides a visual representation of a typical many-to-many RNN structure, illustrating the flow from multiple inputs through hidden states to multiple outputs.

RNNs come in various architectures beyond the standard architectures shown in Figures 3.5–3.7. Training RNNs can also be tricky due to the challenges in calculating gradients across different time steps during backpropagation. This process can lead to instability in the network. For example, the vanishing gradient problem in RNN causes it to forget some learned parameters as time passes.

The exploding gradient problem can exaggerate the output too much when gradients grow uncontrollably as they're propagated backward through time. Also, standard backpropagation doesn't work directly for RNNs because of their recurrent connections. These problems typically arise when working on a very large dataset. To address these issues in RNN, various algorithms such as backpropagation through time (BPTT) and specialized units such as gated recurrent units (GRUs), bidirectional recurrent neural networks (BiRNNs), and LSTM networks are employed.

3.4.3 Backpropagation Through Time

BPTT is a specialized algorithm used in training RNNs. It is an extension of the standard backpropagation algorithm that can handle temporal dependencies present in RNNs. One of the challenges in RNN is that during the training process, the network's error at each step depends

on all previous steps. BPTT addresses this by gradually "unfolding" the network and treating it as a multilayered structure where each layer corresponds to a different time step in the sequence. The backpropagation algorithm is then applied across this temporal structure by calculating gradients across all time steps.

The BPTT algorithm is widely used in various applications that deal with sequential data. Common natural language processing tasks, such as machine translation, speech recognition, and text generation, heavily rely on RNNs trained with BPTT. BPTT is also used to train more advanced recurrent architectures such as LSTM and GRU networks.

3.4.4 Long Short-term Memory Networks

LSTM is a special type of RNN that can mitigate the vanishing and exploding gradient problems when dealing with long sequences of data. LSTM cell consists of several key components: a memory cell, an input gate, a forget gate, and an output gate. The cell state serves as a memory to enable the flow of information across time steps. The gates, on the other hand, control this flow of information into and out of this cell state by using sigmoid and tanh activations. The forget gate decides which information from the previous time step should be discarded from the cell state, the input gate decides which new information should be stored, and the output gate decides what information from the cell state should be used to produce the output. Through cell state updates, LSTMs can retain important information while discarding irrelevant details. This sophisticated structure enables LSTMs to be very effective for long sequence tasks.

In recent years, LSTMs have shown success in various sequence modeling, especially when integrated with attention mechanisms. However, superior transformer-based architectures such as GPT and BERT can capture long-range dependencies more efficiently without the recurrent structure of LSTMs. Therefore, more hybrid models that combine LSTMs with novel techniques are expected to evolve in the near future.

3.4.5 Gated Recurrent Units

GRUs are a type of RNN introduced by Kyunghyun Cho and his colleagues in 2014 [20]. Similar to LSTMs, GRUs are designed to capture long-term dependencies in sequential data while mitigating the vanishing gradient problem. However, GRUs don't have a memory cell and have an efficient structure consisting of two gates (reset and update) compared to LSTM's three gates (input, forget, and output). The update gate decides how much of the previous information needs to be forwarded to the future, and the reset gate decides how much to forget from the past. This mechanism allows GRUs to capture dependencies of different time steps adaptively and handle moderately long sequences. GRUs may not perform well compared to LSTMs on some tasks that need extensive computational resources. Therefore, the choice between GRUs and LSTMs often depends on the type of applications and available computational resources.

3.4.6 Bidirectional RNNs

Bidirectional recurrent neural networks (BiRNNs) are extensions of RNN that Mike Schuster and Kuldip K. Paliwal introduced in 1997 [21]. BiRNNs process the input sequence in both forward and backward directions with two separate hidden layers that are fed forward to the same output layer. This mechanism enables the network to capture both past and future context for the input sequence.

BiRNN architecture is useful for tasks where the entire sequence is available at once (e.g. speech recognition, machine translation, text classification, etc.). The LSTM cells can also be used within a BiRNN architecture by adding the bidirectional data processing capability. Overall, BiRNNs provide more comprehensive information for each time step and perform better where understanding the full sequence of data is important for modeling.

3.5 Applications

Both CNNs and RNNs have widespread applications across numerous domains and industries. CNNs are used to process hierarchically structured images by capturing features. They are highly efficient and scalable in handling large datasets with fewer parameters than fully connected networks. In recent years, CNNs have revolutionized modern computer vision applications and have become a vital technology for image/video analysis tasks such as classification, object detection, segmentation, and facial recognition. It is currently an integral component of social media for filtering inappropriate content, providing suggestions, and image recognition.

In the medical field, CNNs are utilized to analyze images (e.g. X-rays, MRIs, and CT scans) and diagnose diseases. It is widely used in the bioinformatics field for analyzing genetic sequences and predicting the impacts of genetic variations [22–24]. Autonomous vehicles rely on CNNs to interpret visual data for safe navigation, and it is widely utilized in security footage analysis and augmented reality due to its real-time object recognition capability [25]. CNNs have revolutionized satellite imagery analysis by automating tasks that were once tedious and time-consuming, such as identifying and classifying objects like buildings, roads, vehicles, or specific land features, and facilitating disaster forecasting and large-scale predictive modeling [26–28]. In the entertainment industry, CNN is used to enhance graphics and videos, recommend personalized content, moderate content, and generate fantasy characters [29–31]. In robotics, CNNs are extensively used for processing and interpreting visual data, enabling robots to understand and interact with their environment. CNNs facilitate object classification and recognition, allowing robots to identify and categorize objects. They also enhance scene understanding, aiding robots in comprehending complex environments and making decisions based on visual inputs. In robotic grasping, CNNs enable precise manipulation by recognizing and locating objects, determining their orientation, and predicting the optimal grasping points [32–34]. CNNs are used for autonomous navigation of collaborative robots by helping to map and navigate spaces and identifying obstacles, pathways, and landmarks [35, 36]. This technology is also applied in human–robot interaction, where robots use facial and gesture recognition to respond appropriately to human actions and expressions [37, 38]. Overall, the applications of CNN in the modern science and engineering field are numerous, and it can enhance the functionality of many systems by making them more adaptable and efficient for everyday use.

Similar to CNNs, RNNs also have widespread application across various domains. RNNs have shown good performance in time series forecasting, where they can predict future values based on historical data. Applications in this area include disease spread modeling, population growth projections, stock price prediction, and weather forecasting. In the field of text processing, RNNs and their variants excel at understanding context within sentences by considering the sequence of previous words. A notable example is Google's Neural Machine Translation, which employs deep LSTM networks for language translation [39]. Speech recognition is another field in which RNNs have made substantial contributions. They are widely used in virtual assistant technologies such as Google's voice search, Apple's Siri, Amazon's Alexa, and Microsoft's Cortana. These systems can recognize speakers' voices, patterns, and preferences, enabling interactive communication with

users. RNNs have also found applications in gesture recognition, often in combination with CNNs [40]. This technology has potential in virtual reality, gaming, human–robot collaboration, and novel human–computer interaction interfaces. The sentiment analysis using RNNs can extract emotional content or sentiment from written text, news articles, or social media posts. This capability is particularly useful for e-commerce platforms analyzing customer reviews and for brands monitoring their social media engagement.

3.6 Conclusion

As artificial intelligence continues to evolve, the role of advanced neural network architectures is expected to expand significantly in the coming years. The ability of these models to better learn from images, live streams, and sequential data will enable breakthroughs in many fields. Future applications could include smarter healthcare diagnostics, intelligent education platforms, and more sophisticated interaction within smart cities and autonomous systems. Overall, with sustainable investments and proper regulations, this technology can drive further innovations and improve our quality of life.

References

1 Fukushima, K. (1980). Neocognitron: a self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 36 (4): 193–202.

2 Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning representations by back-propagating errors. *Nature* 323 (6088): 533–536.

3 Lecun, Y., Bottou, L., Bengio, Y. et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86 (11): 2278–2324.

4 Krizhevsky, A., Sutskever, I., and Hinton, G.E. (2012). ImageNet classification with deep convolutional neural networks. *Communications of the ACM* 60: 84–90.

5 Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv*: 1409.1556.

6 Szegedy, C., Wei, L., Yangqing, J. et al. (2015). Going deeper with convolutions. *Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* 1–9.

7 He, K., Zhang, X., Ren, S. et al. (2015). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* 770–778.

8 Ronneberger, O., Fischer, P., and Brox, T. (2015). "U-Net: convolutional networks for biomedical image segmentation." Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. Lecture Notes in Computer Science, volume 9351. Springer, Cham. https://doi.org/10.1007/978-3-319-24574-4_28.

9 Redmon, J., Divvala, S., Girshick, R. et al. (2016). You only look once: unified, real-time object detection. *Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* 779–788.

10 He, K., Gkioxari, G., Dollár, P. et al. (2017). Mask R-CNN. *Proceedings IEEE International Conference on Computer Vision (ICCV)* 2980–2988.

11 Tan, M. and Le, Q.V. (2019). EfficientNet: rethinking model scaling for convolutional neural networks. *arXiv*: abs/1905.11946.

12 Dosovitskiy, A., Beyer, L., Kolesnikov, A. et al. (2020). An image is worth 16×16 words: transformers for image recognition at scale. *arXiv*: 2010.11929.

13 Konar, D., Bhattacharyya, S., Panigrahi, B.K. et al. (2017). Chapter 5 – An efficient pure color image denoising using quantum parallel bidirectional self-organizing neural network architecture. In: *Quantum Inspired Computational Intelligence* (ed. S. Bhattacharyya, U. Maulik, and P. Dutta), 149–205. Boston: Morgan Kaufmann.

14 Fang, W., Chen, Y., and Xue, Q. (2021). Survey on research of RNN-based spatio-temporal sequence prediction algorithms. *Journal on Big Data* 3: 97–110.

15 Nath, S., Marie, A., Ellershaw, S. et al. (2022). New meaning for NLP: the trials and tribulations of natural language processing with GPT-3 in ophthalmology. *The British Journal of Ophthalmology* 106 (7): 889–892.

16 Rothman, D. (2021). *Transformers for Natural Language Processing: Build Innovative Deep Neural Network Architectures for NLP with Python*, *PyTorch, TensorFlow, BERT, RoBERTa, and More*. Packt Publishing, Limited.

17 Chandra, A., Tünnermann, L., Löfstedt, T. et al. (2023). Transformer-based deep learning for predicting protein properties in the life sciences. *eLife* 12: e82819.

18 Sinha, S. (2023). *How ChatGPT Could Revolutionize Academia*. https://spectrum.ieee.org/how-chatgpt-could-revolutionize-academia.

19 Smith, M.S. (2023). *Gemini Is Google's Best AI Model Yet, But Who Cares*? https://spectrum.ieee.org/google-gemini.

20 Cho, K., Merrienboer, B., Gulcehre, C. et al. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv*: 1406.1078.

21 Schuster, M. and Paliwal, K.K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45 (11): 2673–2681.

22 Sarvamangala, D.R. and Kulkarni, R.V. (2022) Convolutional neural networks in medical image understanding: a survey. *Evolutionary Intelligence* 15 (1): 1–22.

23 Vardhana, M., Arunkumar, N., Lasrado, S. et al. (2018). Convolutional neural network for bio-medical image segmentation with hardware acceleration. *Cognitive Systems Research* 50: 10–14.

24 Jia, H., Zhang, J., Ma, K. et al. (2024). Application of convolutional neural networks in medical images: a bibliometric analysis. *Quantitative Imaging in Medicine and Surgery* 14 (5): 3501–3518.

25 Song, H.M., Woo, J., and Kim, H.K. (2020). In-vehicle network intrusion detection using deep convolutional neural network. *Vehicular Communications* 21: 100198.

26 Binti Amit, S.N.K. and Aoki, Y. (2017). "Disaster detection from aerial imagery with convolutional neural network." *2017 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC*), Surabaya, Indonesia, 2017, pp. 239–245, https://doi.org/10.1109/KCIC.2017.8228593.

27 Boonyuen, K., Kaewprapha, P., and Srivihok, P. (2018). Daily rainfall forecast model from satellite image using convolution neural network. *Proceedings International Conference on Information Technology (InCIT*) 1–7.

28 Su, J.L., Yeh, C.C., Alkhaleefah, M. et al. (2023). A deep convolutional neural network for building damage evaluation from satellite images. *Proceedings IGARSS 2023–2023 IEEE International Geoscience and Remote Sensing Symposium* 285–288.

29 Gorwa, R., Binns, R., and Katzenbach, C. (2020). Algorithmic content moderation: technical and political challenges in the automation of platform governance. *Big Data & Society* 7 (1): 2053951719897945.

30 Gongane, V.U., Munot, M.V., and Anuse, A.D. (2022). Detection and moderation of detrimental content on social media platforms: current status and future directions. *Social Network Analysis and Mining* 12 (1): 129.

31 Li, Z., Liu, F., Yang, W. et al. (2022). A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems* 33 (12): 6999–7019.

32 Guo, J., Nguyen, H.T., Liu, C. et al. (2023). Convolutional neural network-based robot control for an eye-in-hand camera. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 53 (8): 4764–4775.

33 Kulik, S. and Shtanko, A. (2020). Using convolutional neural networks for recognition of objects varied in appearance in computer vision for intellectual robots. *Procedia Computer Science* 169: 164–167.

34 Tai, L. and Liu, M. (2016). Mobile robots exploration through cnn-based reinforcement learning. *Robotics and Biomimetics* 3 (1): 24.

35 Ran, L., Zhang, Y., Zhang, Q. et al. (2017). Convolutional neural network-based robot navigation using uncalibrated spherical images. *Sensors* 17 (6): 1341.

36 Foroughi, F., Chen, Z., and Wang, J. (2021). A CNN-Based system for mobile robot navigation in indoor environments via visual localization with a small dataset. *World Electric Vehicle Journal* 12 (3): 134.

37 Bamani, E., Nissinman, E., Meir, I. et al. (2023). Ultra-range gesture recognition using a web-camera in human-robot interaction. *Engineering Applications of Artificial Intelligence* 132: 108443.

38 Yu, J., Qin, M., and Zhou, S. (2022). Dynamic gesture recognition based on 2D convolutional neural network and feature fusion. *Scientific Reports* 12 (1): 4345.

39 Wu, Y., Schuster, M., Chen, Z. et al. (2016). Google's neural machine translation system: bridging the gap between human and machine translation. *arXiv*: abs/1609.08144.

40 Toro-Ossaba, A., Jaramillo-Tigreros, J., Tejada, J.C. et al. (2022). LSTM recurrent neural network for hand gesture recognition using EMG signals. *Applied Sciences* 12 (19): 9700.

4

Deep Learning Using PyTorch

4.1 Introduction to PyTorch

PyTorch is an open-source machine learning framework renowned for its flexible and user-friendly interface. It provides tools for performing tensor operations and importing pre-trained deep learning models [1]. PyTorch also has a wide range of tools for building and training customized learning models. PyTorch's history traces back to 2016 when it was first developed by Facebook's AI Research lab (FAIR) [2]. It was initially developed to address the drawbacks of the existing deep learning frameworks in the Python environment. Although it was originally released by Facebook (Meta), since its release, it quickly became a community-driven project with contributions from developers across the world.

The primary competitors of PyTorch are TensorFlow (created by Google), Keras (now integrated with TensorFlow), and MXNet (developed by Apache) [3]. While each framework has its unique advantages, PyTorch has become particularly popular in the research community because of its user-friendly nature and flexibility. In this book, we used PyTorch for all the examples involving deep learning model training and inference. We have shown how to set up the PyTorch framework on Jetson Nano and Raspberry Pi embedded systems in Chapters 5 and 13. This chapter will focus on setting up the PyTorch framework and adding essential software packages on a Windows system.

4.2 Anaconda and PyTorch for Windows System

This section outlines the process of configuring Anaconda and PyTorch for deep learning on a Windows system. We'll utilize Anaconda, a comprehensive Python distribution, that includes numerous pre-installed scientific computing and machine learning libraries. We will use the Spyder Interactive Development Environment (IDE) for Python, which is included with the Anaconda installation. Spyder offers a user-friendly interface for writing, editing, and testing code. The Windows workstation used for this setup has an Intel Core (TM) i9-14900k 3.2 GHz processor, an NVIDIA GeForce RTX 4080 Super GPU, 64 GB of RAM, a 64-bit (×64) system, and a 4-TB hard drive. The system runs on 64-bit Windows 10 Enterprise, offering a suitable platform for deep learning applications.

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

4.2.1 Anaconda Installation and Environment Setup

To install the latest version of Anaconda, visit https://www.anaconda.com/download and download the Anaconda installer for Windows 64-bit (Figure 4.1). There is an option to register and create an account with Anaconda, but we can skip this registration option on the download page.

Once the installer (Anaconda3-2024.06-1-Windows-x86_64) is downloaded, the installation process is quite simple. Click "Next," accept the license agreement, and proceed with the installation by following on-screen guidelines. Ensure that you have administrative privileges to install it for all users. In the advanced installation options, select "Register Anaconda3 as the system Python 3.12" and "Clear the package cache upon completion," then continue as shown in Figure 4.2.

After the Anaconda installation, the required environment variables need to be updated from Windows settings. This option can be accessed by typing "run" application from the search bar and executing the command "control sysdm.cpl," which will launch the "System Properties" window. After that, navigate to the "Advanced" tab and click on "Environment Variables." This will display all the user and system variables. In the system variables section, select "path" and click "Edit" to update the system's path variable. This step is crucial for running Python compilers and interpreters from different directories (Figure 4.3).

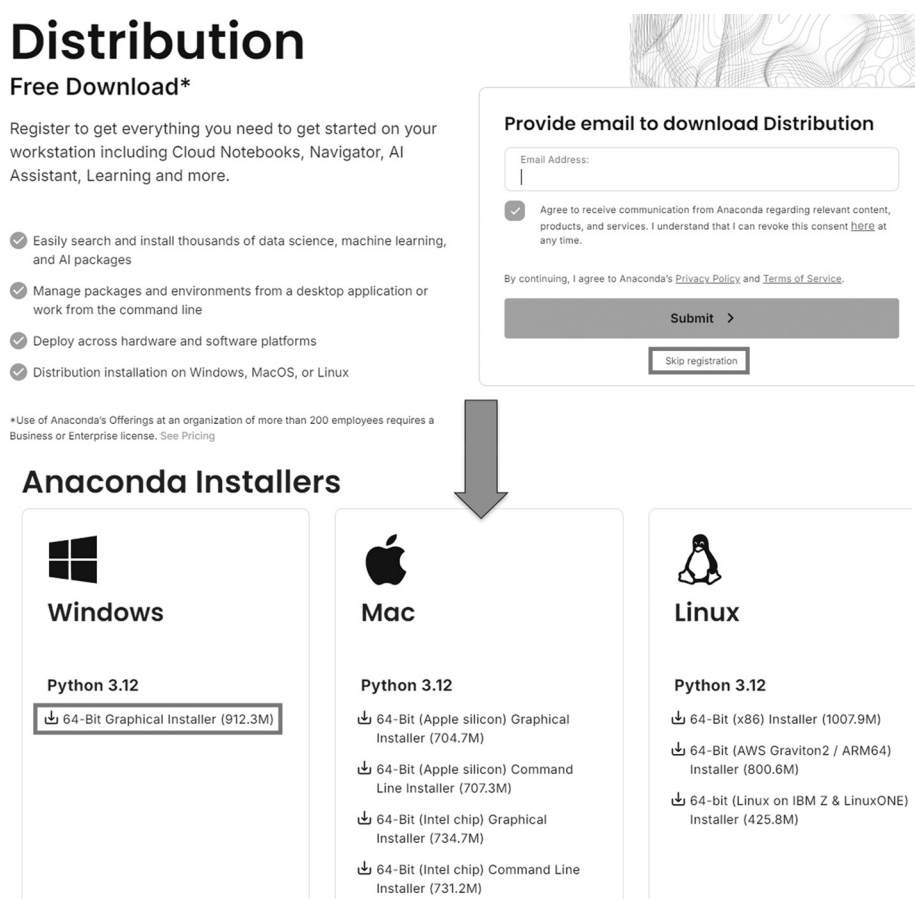


Figure 4.1 Visit https://www.anaconda.com/download and download the Anaconda installer for Windows 64-bit.

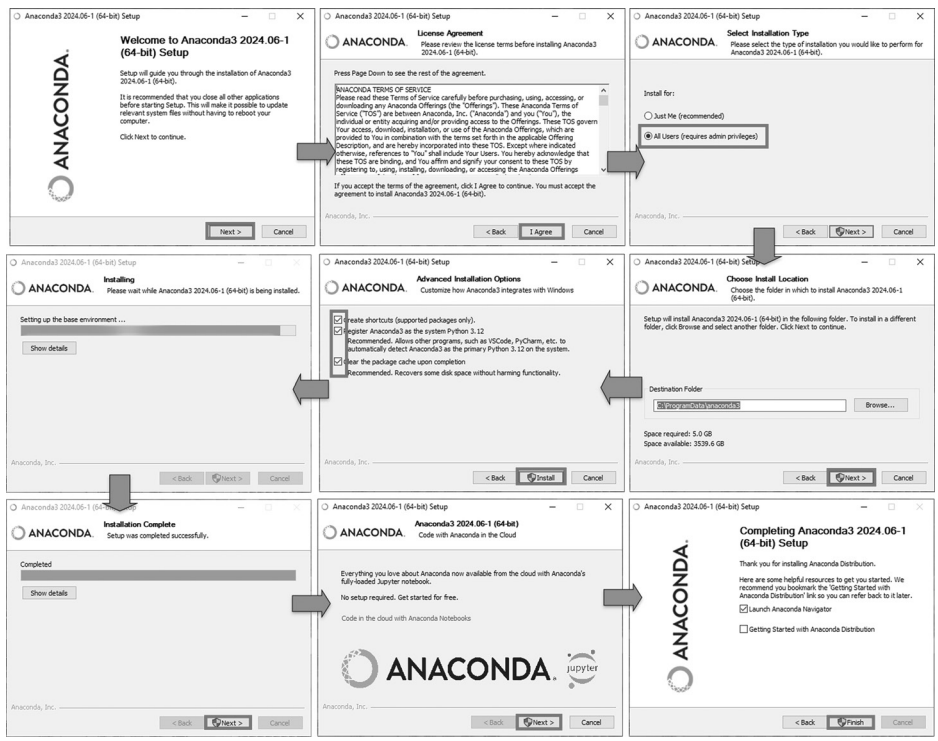


Figure 4.2 Anaconda installation steps for all users using admin privileges. Check the "Register Anaconda3 as the system Python 3.12" and "Clear the package cache upon completion" option during the installation process.

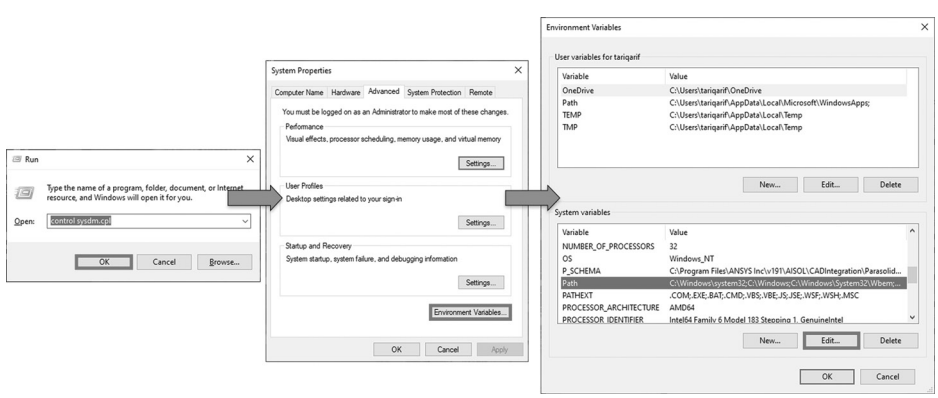


Figure 4.3 Access the system's environment variable path from system properties using "control sysdm.cpl."

Next, we will need to copy the location of the "Scripts" and "Anaconda" folders from the Anaconda installation and add them to the environment variables, as shown in Figure 4.4. By default, these locations are "C:\ProgramData\anaconda3\Scripts" and "C:\ProgramData\anaconda3." After adding these locations, click "OK" in all windows to apply the system settings. If "OK" is not pressed in all the Windows, the settings won't be updated. We may install Anaconda in a different location

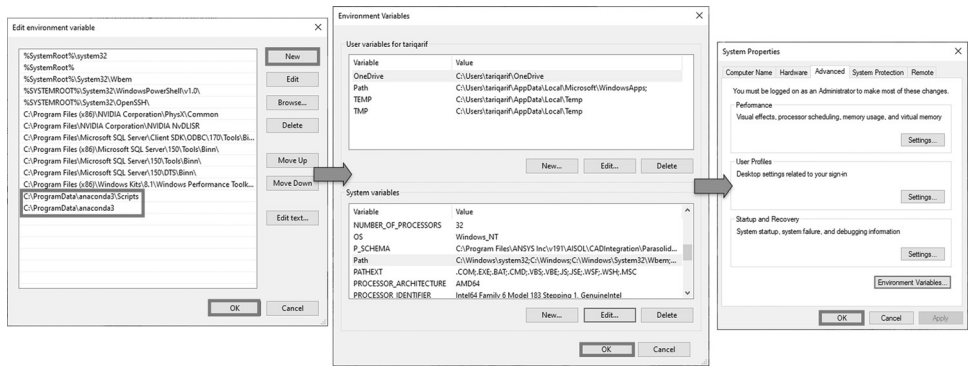


Figure 4.4 Add the installation location of "Scripts" and "Anaconda" folders to the system variables "path."

(see Figure 4.2); in that case, the respective paths need to be copied. Additionally, the "ProgramData" folder in the "C" drive may be hidden by default. If that's the case, we have to enable the "Show hidden items" option from the "View" menu in the Window's folder options.

4.2.2 Setting up the PyTorch Framework

This section presents the installation process for PyTorch and related setup steps. Prior to PyTorch installation, we will need to ensure that the graphics card drivers are up-to-date. If needed, we can check for available updates by accessing "Device Manager" via the Windows search menu and selecting the update option for graphical processing unit (GPU) under device adapters. Also, ensure that the GPU is Compute unified device architecture (CUDA) compatible, which is a parallel computing model designed to enhance its performance. Currently, CUDA support is available for a wide range of NVIDIA GPUs across various product lines, such as NVIDIA's GeForce RTX, Quadro, and Tesla GPUs. A comprehensive list of NVIDIA's CUDA-compatible GPUs is available at [4]. The GPU model used to demonstrate this installation is NVIDIA GeForce RTX 4080 Super, and other hardware configurations are given in Section 4.2.

To install the updated and stable version of PyTorch and its related libraries via the Anaconda package manager, access the corresponding command from https://pytorch.org/get-started/locally/. On this page, we would be able to copy and save the installation command by selecting PyTorch Build (Stable 2.4.0), OS (Windows), package type (Conda), language (Python), and platform (CUDA 12.4) (Figure 4.5).

Next, open the Windows search menu, type "cmd," and select the Command Prompt by right-clicking and choosing "Run as administrator." Navigate to the "Scripts" folder in Anaconda's installation directory using the command "cd C:\ProgramData\anaconda3\Scripts." Once there, activate the root environment by typing activate root. Then, install PyTorch and related libraries by pasting the saved command from Figure 4.5. This step can also be performed using the "Anaconda Prompt."

During installation, a prompt question will ask for permission to proceed with the package updates. Enter "y" to proceed, and it will download and update the required packages. The installation may take between 5 and 30 minutes. Once the process is complete, including preparation, verification, and execution of transactions, we will see the message "done" (Figure 4.6). This indicates that the installation process is completed correctly. However, if any transactions display "failed" and an error message states, "The current user does not have write permissions to the target environment," it means the installation is not successful. This permission error can occur even if the user is an admin

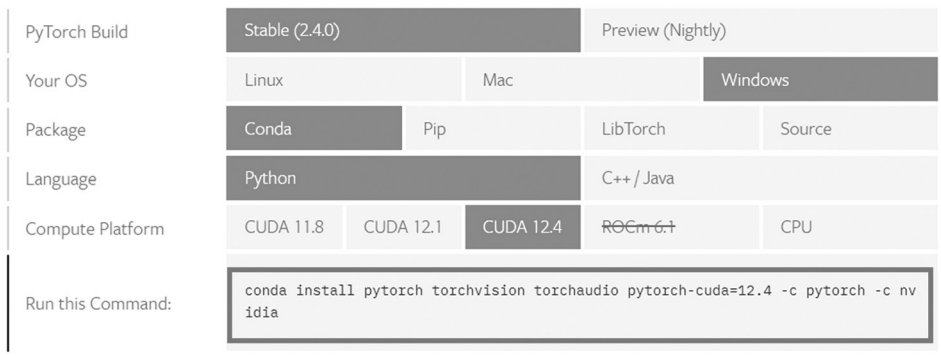


Figure 4.5 Copy the installation command for "PyTorch," "torchvision," and "torchaudio" via the Anaconda package.



Figure 4.6 Launch the command prompt as an administrator, and install PyTorch and related packages.

to the workstation. To fix this issue, navigate to the Anaconda installation folder (e.g. C:\ProgramData\anaconda3), right-click the "anaconda3" folder, and select properties. In the properties window, grant full permissions to the domain user (or the current user) by checking all the "Allow" options and clicking "OK" (Figure 4.7).

After granting full permissions, previously installed PyTorch should be uninstalled from the control panel's "Programs and Features" or using any uninstaller applications such as "Revo Uninstaller Pro" (https://www.revouninstaller.com/products/revo-uninstaller-free). It is highly recommended to use "Revo Uninstaller Pro" as it will clean Window's registry entries from the previous installation. Once uninstalled, PyTorch can be installed using the command prompt, as shown in Figures 4.5 and 4.6.

After successfully installing PyTorch, we can launch the Spyder IDE within an Anaconda environment. To do this, open the "Anaconda Prompt" as an administrator via the Windows start menu and enter the command "spyder" (Figure 4.7). If we launch Spyder directly from the start menu, there might be issues with Python locating CUDA on the workstation. This issue can vary depending on the user's security and permission levels.

In the Spyder IDE's console, we can verify PyTorch installation by importing the "torch" library and entering the "torch.cuda.is_available()" command, as shown in Figure 4.8.

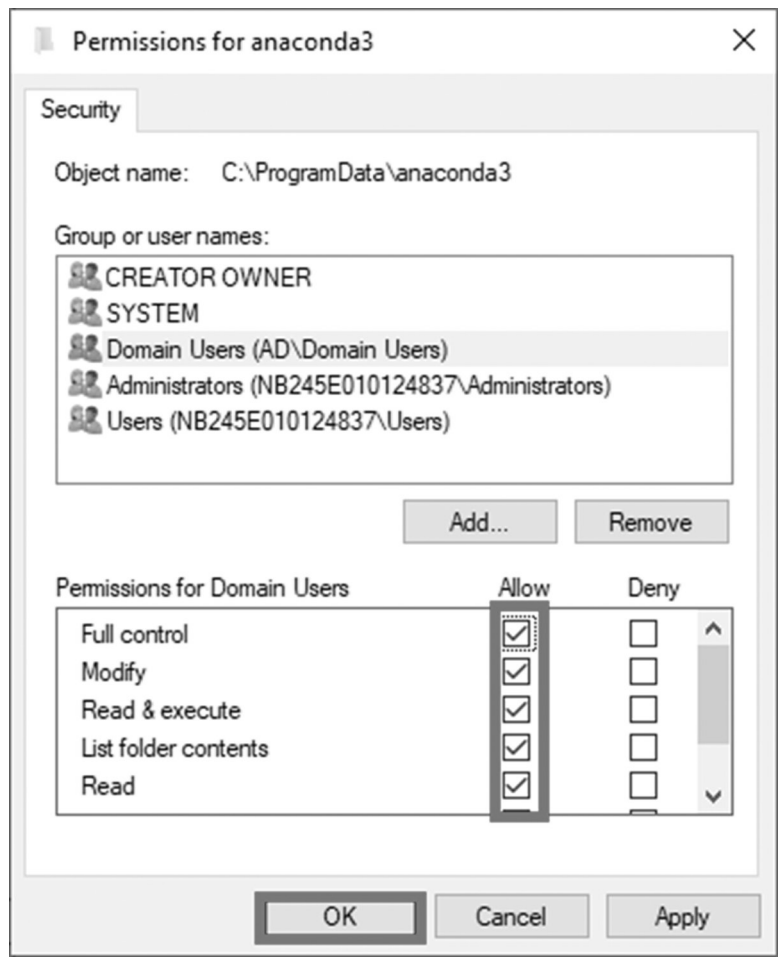


Figure 4.7 Update the "anaconda3" folder permission of the user in case of a permission error.

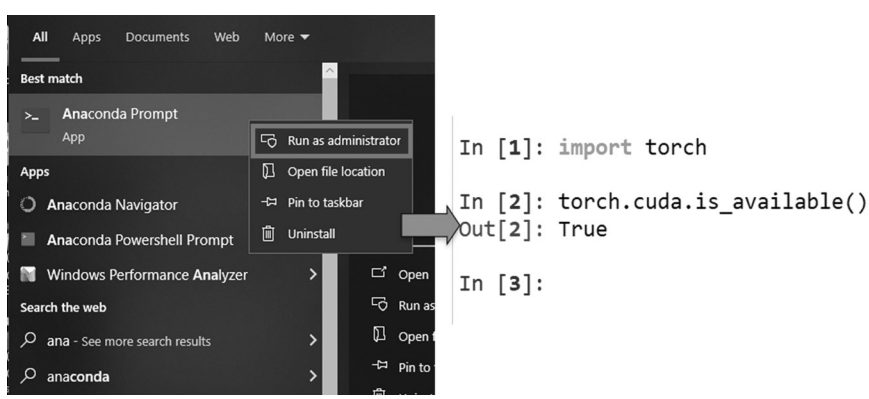


Figure 4.8 Run Spyder using Anaconda Prompt and verify CUDA availability.

4.2.2.1 Visual Studio Professional

Visual Studio is an IDE by Microsoft that supports various software development platforms and programming languages. Installation of Visual Studio is necessary for deep learning computation, as the CUDA toolkit includes many components that are dependent on Microsoft Visual C++ compilers and libraries. Visual Studio also provides build tools and SDKs that CUDA uses to create and compile GPU-accelerated codes. With the help of Visual Studio, developers can leverage the full power of NVIDIA GPUs in a robust development environment and can enhance computational efficiency. For many CUDA versions, specific Visual Studio versions are recommended to ensure compatibility and optimal performance. We can verify the compatible Visual Studio version for the CUDA version we need to install. In Figures 4.11 and 4.12, we can see from the installer that the updated version of CUDA is 12.6. Thus, we can verify from [5] that both Visual Studio 2022 and 2019 are compatible with CUDA 12.6 (Figure 4.9).

We can download the Visual Studio Professional 2022 installer (VisualStudioSetup.exe) from https://visualstudio.microsoft.com/vs/ and initiate the installation by double-clicking the file. Once the installer is launched, choose ".NET desktop development" and "Desktop development with C++," then click "install" to include these packages with the core Visual Studio installation (Figure 4.10).



Figure 4.9 Verify the required Visual Studio version for the corresponding CUDA version.

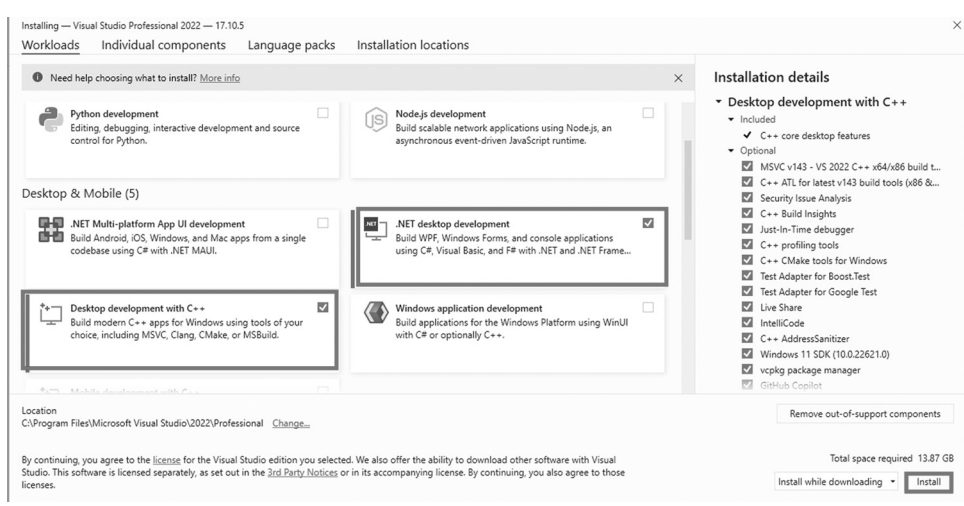


Figure 4.10 During the installation process of Visual Studio 2022, include ".NET desktop development" and "Desktop development with C++" packages.

4.2.2.2 CUDA and cuDNN Installation

CUDA and CUDA deep neural network library (cuDNN) are both developed by NVIDIA, and they serve different aspects of GPU-accelerated computing. CUDA is a parallel computing platform and application programming interface (API) that helps developers utilize NVIDIA GPUs. On the other hand, cuDNN is an optimized deep learning library built on top of CUDA.

After installing Visual Studio Professional 2022, we will need to install CUDA and cuDNN. The CUDA toolkit installer can be downloaded from https://developer.nvidia.com/cuda-downloads. During the download process, we can select the operating system, appropriate architecture, and Windows version, as shown in Figure 4.11. For our demonstration, we downloaded the ".exe (local)" installer file.

After downloading the NVIDIA CUDA toolkit installer (cuda_12.6.0_560.76_windows.exe), we can launch it from the download location. The installation process is straightforward and involves agreeing to the license agreement and clicking "Next" through the prompts. For visual guidance, Figure 4.12 provides screenshots illustrating key steps of this installation process.

Next, the cuDNN installer can be downloaded for specific CUDA version. First, verify the appropriate version using the support matrix available at [5].

Here, we can see that for CUDA 12.x toolkit, cuDNN 9.3.0 can be used. To download this, select the target platform's operating system, architecture, and Windows version from [6], and download the installer "cudnn_9.3.0_windows.exe" (Figure 4.13).

The installation of cuDNN using the "cudnn_9.3.0_windows.exe" installer is very straightforward and involves agreeing to the license agreement and clicking "Next" through the prompts. For visual guidance, these installation steps are given in Figure 4.14.

To verify the installations of CUDA and cuDNN, we can launch the Spyder IDE and import "torch" from the console to check their versions. As shown in Figure 4.15, the installed torch version is 2.4.0. The device used is specified as "0," and by using the command "torch.cuda.get_device_name(0)", we can identify the GPU name in the workstation. Also, the command "torch.version.cuda" returns the CUDA version, which is 12.4, and "torch.backends.cudnn.version()" returns the cuDNN version, which is 90100.

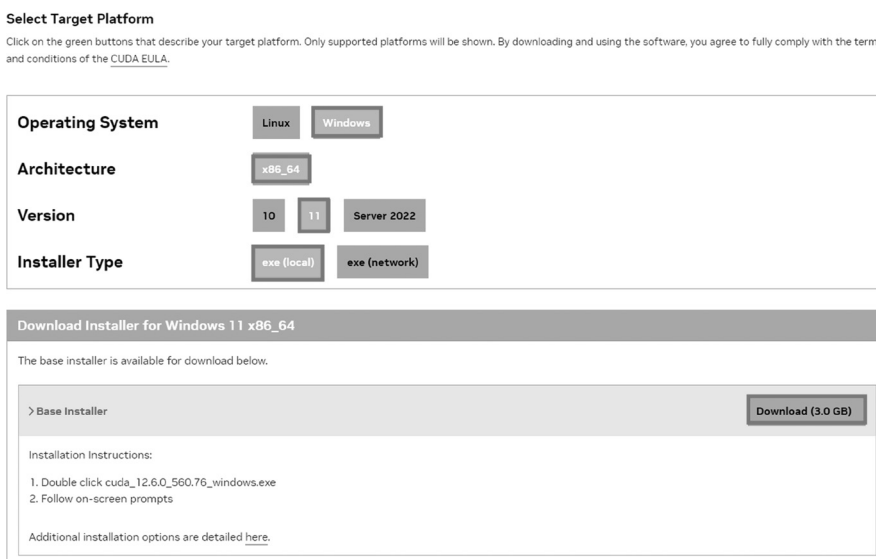


Figure 4.11 Download the CUDA installer by selecting the operating system, architecture, and Windows version.

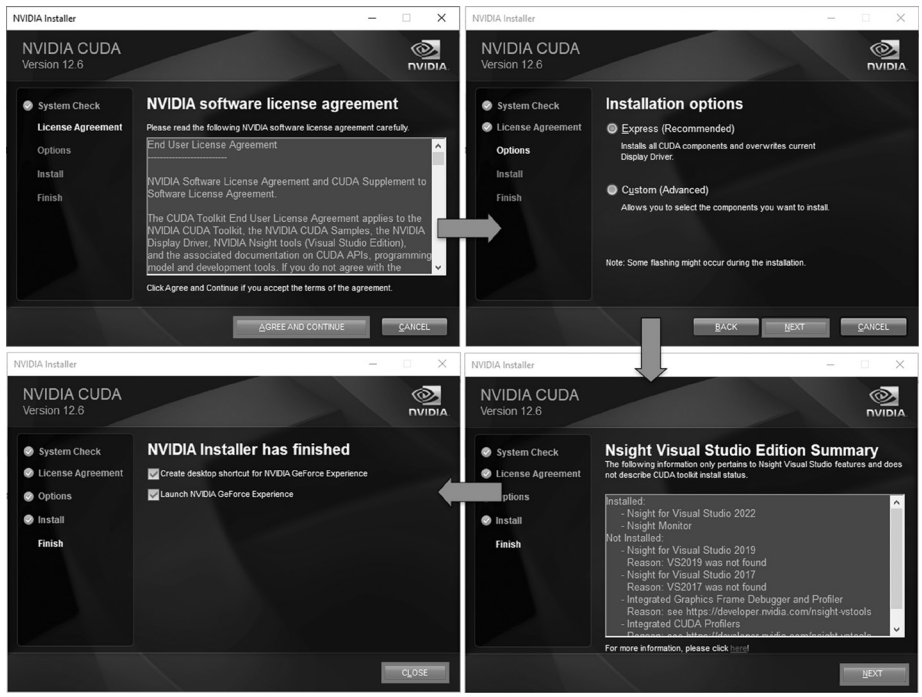


Figure 4.12 Key steps for installing CUDA by launching the "cuda_12.1.1_531.14_windows.exe" installer.

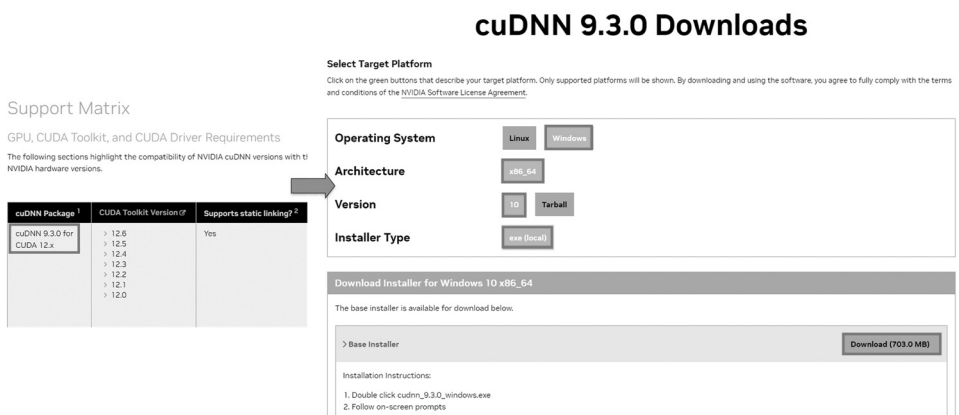


Figure 4.13 Check the cuDNN version for CUDA 12.x and download the installer by selecting the operating system, architecture, and Windows version.



Figure 4.14 Key steps for installing cuDNN by launching the "cudnn_9.3.0_windows.exe" installer.

```
In [1]: import torch

In [2]: torch.__version__
Out[2]: '2.4.0'

In [3]: torch.cuda.current_device()
Out[3]: 0

In [4]: torch.cuda.get_device_name(0)
Out[4]: 'NVIDIA GeForce RTX 4080 SUPER'

In [5]: torch.version.cuda
Out[5]: '12.4'

In [6]: torch.backends.cudnn.version()
Out[6]: 90100
```

Figure 4.15 Import "torch," check device name, CUDA version, and cuDNN version from Spyder IDE.

4.3 Other Essential Packages for Deep Learning

In this section, we will demonstrate some common libraries that need to be installed in the workstation to support image processing, computer vision, and deep learning tasks. The first one is Open Source Computer Vision (OpenCV) library. It is highly useful for image and video frame manipulation during machine and deep learning applications. To install OpenCV, open "Anaconda Command Prompt" from the Windows search menu and launch it as an administrator. The "Command Prompt" terminal typically runs from the user's profile (e.g. C:\Users\username). However, it will run from the "system32" directory of Windows if opened as administrator (e.g. C:\WINDOWS\system32). From the "system32" directory, change the directory location to the "Scripts" folder of the Anaconda installation using the command "cd" and "Scripts" folder location (Figure 4.16). In this location, update the "pip" package installer to the latest version by running the command "python -m pip install --upgrade pip" (Figure 4.16). Next, install Python's OpenCV library using the command "pip install opencv-python." This will download and install the latest version of OpenCV, and it will also install NumPy as a dependency.

Next, install the image augmentation library called "imgaug" [7]. This library is essential in deep learning applications as it can artificially increase the number of training images by applying various image processing and augmentation techniques. To install this library using "pip," use the "pip install imgaug" in the Anaconda command prompt. Ensure that this command is executed from the "Scripts" folder of the Anaconda installation as an administrator (Figure 4.16). This ensures that the library is correctly integrated with the Anaconda environment.

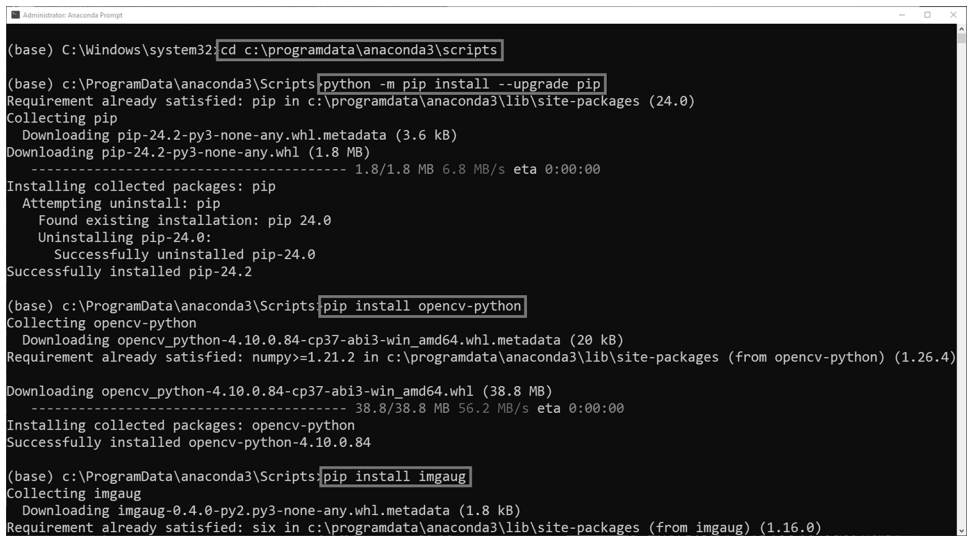


Figure 4.16 Access to the "scripts" folder in the Anaconda command prompt, update "pip," and install "OpenCV" and "imgaug" libraries.

We will also need to install PyTorch's "pretrainedmodels" to access pre-trained convolutional networks with unique interfaces or APIs. Although PyTorch's "torchvision" library already provides several built-in model architectures with pre-trained weights for various datasets, "pretrainedmodels" library further expands the selection of models within the PyTorch ecosystem. To install this package within the Anaconda environment, open the Anaconda command prompt as administrator and execute the command "pip install pretrainedmodels." Figure 4.17 shows this installation process.

To monitor real-time GPU information, such as memory utilization, temperature, power consumption, and other essential configurations and settings, we can use the "nvidia-smi" command from the Command Prompt or Anaconda Command Prompt (Figure 4.17). The current workstation, which

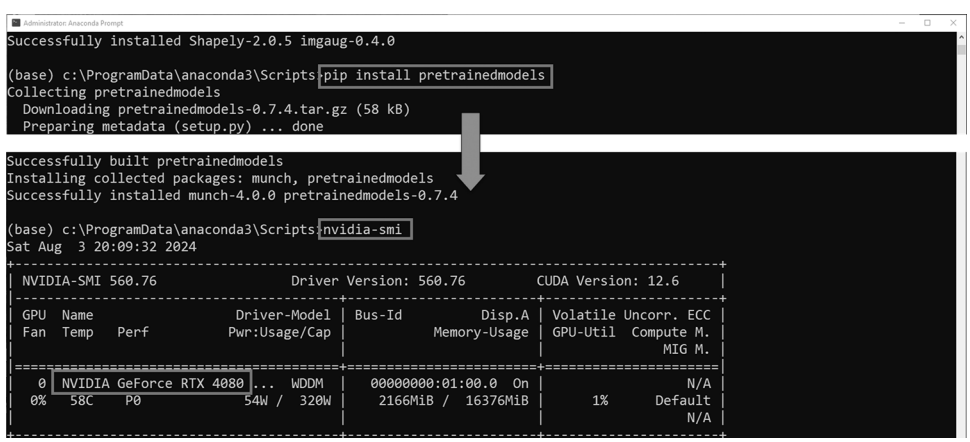


Figure 4.17 Install "pretrainedmodels" for adding more model architectures within the PyTorch ecosystem and verify GPU configuration using "nvidia-smi."

demonstrates the setup of PyTorch and related packages, is equipped with a single NVIDIA GeForce RTX 4080 Super GPU. For a workstation with two GPUs, the setup process remains the same, but during Python programming, we need to implement data parallelism to utilize multiple GPUs.

4.4 Introduction to Tensor

The idea of tensors started in the mid-1800s, and it has been developed with the help of many mathematicians and physicists. Gauss and Riemann's work on differential geometry set the stage for tensors, while Ricci-Curbastro and Levi-Civita formalized tensor calculus in the early 1900s [8–10]. These pioneers established the groundwork for tensors, leading to their extensive use in modern deep learning models.

4.4.1 Defining a Tensor

A tensor is a multidimensional mathematical object or array of numbers that generalizes the concept of scalars, vectors, and matrices to higher dimensions. Unlike matrices, which are always two-dimensional (2D) tensors, tensors can have any number of dimensions. For example, a scalar is a single numerical value representing a zero-dimensional tensor. A vector expands this concept to one dimension, where an ordered list of numbers is a one-dimensional tensor. The one-dimensional tensor can represent a point or direction in space. A matrix further expands this concept into 2D, which can be a rectangular array of numbers organized into rows and columns. The 2D tensor is useful for representing linear transformations and systems of equations. Beyond 2D (n-dimensions), a tensor can accommodate higher-dimensional data structures using an array of numbers. This concept is essential in many scientific fields, such as mathematics, physics, computer graphics, and deep learning, as they require the representation of complex relationships and multidimensional data. Figure 4.18 shows a representation of 0D to 4D tensors using cubic mathematical objects.

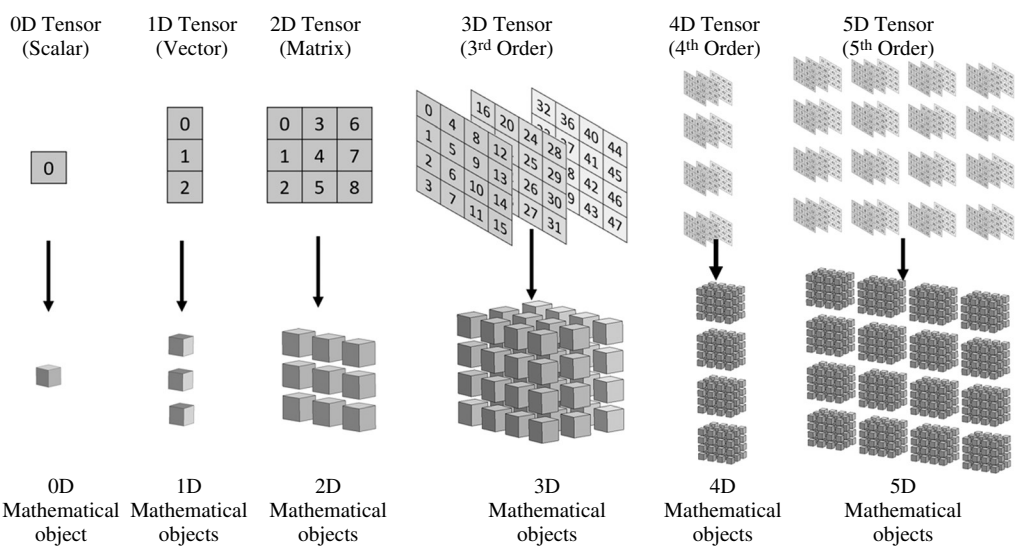


Figure 4.18 Illustration of 0D to 4D tensors using cubic mathematical objects.

```
1  import torch
2  # define a 2D tensor from using list data
3  data1 = [[1, 2, 3], [4, 5, 6],[7, 8, 9]]
4  mytensor = torch.tensor(data1)
5  print(mytensor)
6  print('tensor shape:',mytensor.shape)
7  print('tensor dimension:',mytensor.dim())
```

Output

```
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
tensor shape: torch.Size([3, 3])
tensor dimension: 2
```

Figure 4.19 Create a 2D tensor with 3 by 3 size using PyTorch.

```
9  # define a 3D tensor
10 data2 = [[[1, 2, 3]], [[4, 5, 6]],[[7, 8, 9]]]
11 mytensor = torch.tensor(data2)
12 print(mytensor)
13 print('tensor shape:',mytensor.shape)
14 print('tensor dimension:',mytensor.dim())
```

```
tensor([[[1, 2, 3]],

        [[4, 5, 6]],

        [[7, 8, 9]]])
tensor shape: torch.Size([3, 1, 3])
tensor dimension: 3
```

Figure 4.20 Creating a 3D tensor with 3 by 1 by 3 size using PyTorch.

4.4.2 Tensors in PyTorch

In PyTorch, a tensor is the fundamental data structure for computation that represents multi-dimensional arrays. It is very similar to NumPy arrays but with added capabilities such as automatic differentiation, flexible operations, and GPU acceleration. Tensors in PyTorch can have many dimensions (0D to n-D) and are highly effective in handling a wide range of data types (float32, float64, integers, complex numbers, etc.).

To create a tensor in PyTorch, we first need to import the torch library. Suppose we want to generate a 3 by 3, 2D tensor. In that case, the codes shown in Figure 4.19 can be used in Spyder IDE to create the tensor and display its shape and dimensions.

Now, suppose we want to create a 3D tensor with dimensions of 3 by 1 by 3. This time we have to use extra square brackets to indicate the completion of a 2D array. In this case, we use the same set of data but stack each 3 by 1, 2D array together to form a 3D array. Figure 4.20 shows the code for creating this tensor and checking its shape and dimensions.

4.5 Basic Torch Operations in PyTorch

In this section, we'll explore how to create random and zero tensors using PyTorch, and perform basic mathematical operations such as multiplication and division. Random tensors play a significant role in deep learning, particularly for initializing the weights and biases of a neural network, regularization and dropout, and data augmentation. Proper tensor initialization can greatly influence the training process by promoting diverse feature learning. Additionally, if all weights were set to the same value, the gradients during backpropagation would be identical for all units, which may lead to ineffective learning. Similar to random tensors, sometimes we will need to create a tensor of zeros for deep learning. Most of the time, in neural networks, biases are initialized to zero to ensure that they do not introduce any initial preferences. Zero tensors are also used for padding to maintain the spatial dimensions of the input, for masking to exclude specific elements during

computations, and in initializing the hidden states of recurrent neural networks to establish a starting point for sequences. Figure 4.21 shows how "torch.rand" can be utilized to create a tensor with random numbers of a specified size and how "torch.zeros" can be used to generate tensors filled with zeros of a specified size.

Figure 4.22 shows how to create two random 2D tensors and perform element-wise multiplication on them. This kind of element-by-element tensor multiplication is frequently used in applying activation functions, attention mechanisms, and cost functions.

Figure 4.23 shows how to perform element-wise division on the random tensors created in Figure 4.21. This kind of element-by-element division of tensors is frequently used in applying normalization, regularization, inverse operations, and gradient scaling. Deep learning may also require matrix multiplication that combines elements across rows and columns during layer transformation operations in fully connected layers, convolutional layers, and backpropagation. Figure 4.23 also shows the matrix multiplication operation of two tensors using the "torch.matmul" function.

```
16  # define a tensor using random number
17  mytensor_random = torch.rand((2, 3))
18  print(mytensor_random)
19
20  # define a tensor with zeros
21  mytensor_zero = torch.zeros((2, 3))
22  print(mytensor_zero)
```

```
tensor([[0.8632, 0.8295, 0.0532],
        [0.2680, 0.5959, 0.5066]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

Figure 4.21 Use "torch.rand" and "torch.zeros" to generate random tensor and zero tensor in PyTorch.

```
24  # define two tensors using random number
25  mytensor1 = torch.rand((3, 3))
26  print('mytensor1: \n',mytensor1)
27  mytensor2 = torch.rand((3, 3))
28  print('mytensor2: \n',mytensor2)
29
30  # element-wise multiplication
31  multi_tensor = mytensor1 * mytensor2
32  print('mulitiplication: \n',multi_tensor)
```

```
mytensor1:
 tensor([[0.1522, 0.2610, 0.9218],
        [0.8155, 0.6534, 0.3695],
        [0.8540, 0.2709, 0.5154]])
mytensor2:
 tensor([[0.0353, 0.9446, 0.8975],
        [0.5812, 0.4730, 0.9538],
        [0.7366, 0.9606, 0.2431]])
mulitiplication:
 tensor([[0.0054, 0.2466, 0.8272],
        [0.4740, 0.3091, 0.3524],
        [0.6291, 0.2602, 0.1253]])
```

Figure 4.22 Create two random tensors and perform an element-wise multiplication operation on them.

```
# element-wise division
div_tensor = mytensor1/mytensor2
print('division: \n',div_tensor)

# matrix multiplicaton
multi_matrix = torch.matmul(mytensor1,mytensor2)
print('matrix multiplication: \n',multi_matrix)
```

```
division:
 tensor([[4.3119, 0.2763, 1.0271],
        [1.4033, 1.3815, 0.3874],
        [1.1593, 0.2820, 2.1199]])
matrix multiplication:
 tensor([[0.8361, 1.1527, 0.6096],
        [0.6807, 1.4344, 1.4450],
        [0.5672, 1.4299, 1.1501]])
```

Figure 4.23 Perform element-wise division and matrix multiplication of tensors.

4.6 Gradient Calculation in PyTorch

The gradient calculation in deep learning is a very important operation that is primarily used for optimizing neural network parameters. The gradient indicates how much a small change in weights will affect the loss function during training operation. By using these gradients, optimization algorithms such as stochastic gradient descent (SGD) can adjust the weights in the direction that reduces the loss. This process is also known as backpropagation, and it allows the network to learn from its mistakes and gradually refine its predictions through weight parameters. Figure 4.24 shows how we can utilize PyTorch to calculate the gradient of a 0D tensor. Here, a tensor "x" is created with an initial value of 10.0. The "requires_grad=True" in PyTorch is used to track all operations on "x" so that it can compute gradients later. A function, $y = x^3 + 5x$ is defined for which we want to calculate the gradient. Gradient computation through "y.backward()" generates the derivative is $\frac{dy}{dx} = 3x^2 + 5$. After that, if we access the computed gradient using "x.grad," our final value becomes $\frac{dy}{dx} = 3*(10)^2 + 5 = 305$ (Figure 4.24).

Figure 4.25 shows the gradient calculation for a 2D tensor. Here, a 2D tensor "x" is created with an initial value of

$$x = [[2.0\ 1.5], [1.0\ 3.0]]$$

The "requires_grad=True" is used to track all operations on "x" so that it can compute gradients later. For the function, $y = x^3 + 5x$, the value of y is

$$y = [[18\ 10.875], [6.0\ 42.0]]$$

```
import torch
# Create a tensor and enable gradient calculation
x = torch.tensor(10.0, requires_grad=True)
# Define a function y = x^3 + 5x
y = x**3 + 5*x
# Compute the gradient
y.backward()
# Print the gradient (dy/dx)
print(f"The gradient of y with respect to x is: {x.grad}")
```

```
The gradient of y with respect to x is: 305.0
```

Figure 4.24 Gradient computation example of a 0D tensor using PyTorch.

```
import torch
# Create a 2D tensor and enable gradient calculation
x = torch.tensor([[2.0, 1.5], [1.0, 3.0]], requires_grad=True)
print('tensor x: \n',x)
# Define a function y = x^3 + 5x
y = x**3 + 5*x
print('value of y: \n',y)
# reduce y to a scalar
scalar_y = y.mean()
print('scalar y: \n',scalar_y)
# Compute the gradient
scalar_y.backward()
# Print the gradient (dy/dx)
print(f"The gradient of y with respect to x is: \n{x.grad}")
```

Output

```
tensor x:
 tensor([[2.0000, 1.5000],
        [1.0000, 3.0000]], requires_grad=True)
value of y:
 tensor([[18.0000, 10.8750],
        [ 6.0000, 42.0000]], grad_fn=<AddBackward0>)
scalar y:
 tensor(19.2188, grad_fn=<MeanBackward0>)
The gradient of y with respect to x is:
tensor([[4.2500, 2.9375],
        [2.0000, 8.0000]])
```

Figure 4.25 Gradient calculation of a 2D tensor using the average of all elements in the tensor.

We have to reduce the value of "y" to a scalar so that we can call the "backward()" function. To create a scalar, we can apply various reduction operations. For example, calculating the mean value using "y.mean(),"calculating the sum using "y.sum(),"finding the index of the maximum value using "y.max(),"index of the minimum value using "y.mean(),"etc.

In Figure 4.25, we calculate the mean of all elements in the tensor. The scalar value of "y" becomes

$$scalar_y = \frac{18.0 + 10.875 + 6.0 + 42.0}{4} = 19.2188.$$

However, since the mean operation here also averages out the contributions of all elements, the gradient of each element will be $\frac{1}{number\ of\ elements}$ i.e. ¼ = 0.25. After gradient computation through "y.backward()" the derivative is $\frac{dy}{dx} = 3x^2 + 5$, and we find the final gradient of x is

$$\frac{dy}{dx} = [[0.25\ 0.25],\ [0.25\ 0.25]] \ * \ [[17.0\ 11.75],\ [8.0\ 32.0]] = [[4.25\ 2.9375],\ [2.0\ 8.0]]$$

where $3x^2 + 5$ *for location* [0,0] is $3*2.0^2 + 5 = 17$

$3x^2 + 5$ *for location* [0,1] is $3*1.5^2 + 5 = 11.75$

$3x^2 + 5$ *for location* [1,0] is $3*1.0^2 + 5 = 8.0$

$3x^2 + 5$ *for location* [1,1] is $3*3.0^2 + 5 = 32.0$

Figure 4.26 shows the gradient calculation using "y.sum()" for creating a scalar value. The ".sum()" adds all the elements of the tensor to produce a scalar value. In that case, since the output is the sum of all elements in the tensor, each element has an equal contribution, and the gradient of each element is 1. The scalar value of y becomes 18.0 + 10.875 + 6.0 + 42.0 = 76.875. After gradient computation through "y.backward()," the derivative is $\frac{dy}{dx} = 3x^2 + 5$, and we find the final gradient of x is

$$\frac{dy}{dx} = [[1.0\ 1.0],\ [1.0\ 1.0]] \ * \ [[17.0\ 11.75],\ [8.0\ 32.0]] = [[17.0\ 11.75],\ [8.0\ 32.0]]$$

Besides ".mean()" and ".sum()," various other functions are also used for scaling the gradient. Figure 4.27 shows the effect of using ".max()" and ".min()" to calculate the scalar value of "y" when the same 2D tensor (Figure 4.25) is used. When ".max()" is used for backpropagation, only the elements with the maximum value will have non-zero gradients, and it will fetch the index of the maximum value. In this example tensor, it is the location [1,1]. ".max()" is typically utilized for

```
1  import torch
2  # Create a 2D tensor and enable gradient calculation
3  x = torch.tensor([[2.0, 1.5], [1.0, 3.0]], requires_grad=True)
4  print('tensor x: \n',x)
5  # Define a function y = x^3 + 5x
6  y = x**3 + 5*x
7  print('value of y: \n',y)
8  # reduce y to a scalar
9  scalar_y = y.sum()
10 print('scalar y: \n',scalar_y)
11 # Compute the gradient
12 scalar_y.backward()
13 # Print the gradient (dy/dx)
14 print(f"The gradient of y with respect to x is: \n{x.grad}")
```

Output

```
tensor x:
 tensor([[2.0000, 1.5000],
        [1.0000, 3.0000]], requires_grad=True)
value of y:
 tensor([[18.0000, 10.8750],
        [ 6.0000, 42.0000]], grad_fn=<AddBackward0>)
scalar y:
 tensor(76.8750, grad_fn=<SumBackward0>)
The gradient of y with respect to x is:
tensor([[17.0000, 11.7500],
        [ 8.0000, 32.0000]])
```

Figure 4.26 Gradient calculation of a 2D tensor using the total contribution of each element to the sum.

Output Using 'y.max()'

```
tensor x:
 tensor([[2.0000, 1.5000],
        [1.0000, 3.0000]], requires_grad=True)
value of y:
 tensor([[18.0000, 10.8750],
        [ 6.0000, 42.0000]], grad_fn=<AddBackward0>)
scalar y:
 tensor(42., grad_fn=<MaxBackward1>)
The gradient of y with respect to x is:
tensor([[ 0.,  0.],
        [ 0., 32.]])
```

Output Using 'y.min()'

```
tensor x:
 tensor([[2.0000, 1.5000],
        [1.0000, 3.0000]], requires_grad=True)
value of y:
 tensor([[18.0000, 10.8750],
        [ 6.0000, 42.0000]], grad_fn=<AddBackward0>)
scalar y:
 tensor(6., grad_fn=<MinBackward1>)
The gradient of y with respect to x is:
tensor([[0., 0.],
        [8., 0.]])
```

Figure 4.27 Gradient calculation of a 2D tensor using the index of maximum and minimum values.

reinforcement learning and in some decision-making models. When the ".min()" is used for back-propagation, elements with the minimum value will have non-zero gradients, and it will fetch the index of the minimum value. ".min()" is useful for some optimization tasks.

4.7 Exercise Problems

(Q1) Sections 4.2 and 4.3 of this chapter provide guidelines for installing the PyTorch framework and necessary packages on a Windows workstation with an NVIDIA GPU. Use these instructions to configure a workstation similar to the one described in this chapter for deep learning applications.

(Q2) A 2D tensor, "x" is defined below:

$$x = [[2.0\ 4.0],\ [1.0\ 9.0]]$$

If the function "y" is defined as $y = x^2 + 4x + 1$, find the gradient of y with respect to x using the L2 norm.

[Hints: Use ".norm()" to reduce "y" to a scalar. By default, ".norm()" uses the L2 norm, which computes the vector norm of the tensor. This process is also known as L2 regularization. Use Python and PyTorch.]

(Q3) Two 3D tensors are defined as

$$x1 = [[[1.5, 0],\ [1.0, 0.8]],\quad [[0.5, 0.6],\ [1.0, 0]],\quad [[1.2, 2.0],\ [0, 1.0]]]$$

$$x2 = [[[1.0, 1.0],\ [2.0, 1.0]],\quad [[3.2, 0],\ [1.5, 1.5]],\quad [[1.5, 0],\ [1.0, 0.4]]]$$

Perform element-wise multiplication of these two tensors using PyTorch and find the gradient of the resultant tensor "x" using only the index of maximum value. Here the function "y" is defined as $y = x^3 + 2x^2 + 3$,

[Hints: Use ".max()" for backpropagation. After multiplying two tensors, use "clone().detach(). requires_grad_(True)" for gradient calculation. This will detach the new resultant tensor from the original computation graph. Use Python and Pytorch.]

References

1 Paszke, A., Gross, S., Massa, F. et al. (2019). PyTorch: an imperative style, high-performance deep learning library. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Article 721. Curran Associates Inc.

2 Yann LeCun, J.P. and Schroepfer, M. (2018). *FAIR Turns Five: What We've Accomplished and Where We're Headed*. https://engineering.fb.com/2018/12/05/ai-research/fair-fifth-anniversary.

3 Cohen, A. (2023). *A Comparative Analysis of TensorFlow, PyTorch, MXNet, and scikit-learn*. https://iamitcohen.medium.com/a-comparative-analysis-of-tensorflow-pytorch-mxnet-and-scikit-learn-2072fe566df7.

4 NVIDIA Developer (2024). *Your GPU Compute Capability*. https://developer.nvidia.com/cuda-gpus.

5 NVIDIA (2024). *Support Matrix - GPU, CUDA Toolkit, and CUDA Driver Requirements*. https://docs.nvidia.com/deeplearning/cudnn/latest/reference/support-matrix.html.

6 NVIDIA (2024). *cuDNN 9.3.0 Downloads*. https://developer.nvidia.com/cudnn-downloads.

7 Others, A.J.A. (2020). *imgaug*. https://github.com/aleju/imgaug.

8 Iosifidis, D. (2021). Riemann tensor and Gauss–Bonnet density in metric-affine cosmology. *Classical and Quantum Gravity* 38 (19): 195028.

9 Charmousis, C., Copeland, E.J., Padilla, A. et al. (2012). General second-order scalar-tensor theory and self-tuning. *Physical Review Letters* 108 (5): 051101.

10 Toscano, F. (2017). An Italian calculus for general relativity. *Lettera Matematica PRISTEM* 5 (1): 25–31.

5

Introduction to Jetson Nano and Setup

5.1 Introduction to Jetson Embedded Devices

NVIDIA's Jetson platform represents a significant advancement in embedded systems, enabling low-cost artificial intelligence (AI) applications in robotics and image processing tasks. NVIDIA started this development in 2014 with the release of the Jetson TK1, which featured a mobile processor, Tegra K1, with a 192-core Kepler GPU [1]. This module brought desktop-class computing performance to embedded systems and enabled substantial parallel computing or computer vision capabilities in a compact, low-power platform. After the TK1, the Jetson TX1 was introduced in 2015, utilizing a more powerful Tegra X1 processor with a Maxwell-based GPU, aiming for AI applications, such as autonomous navigation and deep learning-based inference [2]. In 2017, NVIDIA released Jetson TX2, a credit card-sized embedded platform with a Pascal-based GPU that offers improved processing power and energy efficiency [3]. The Jetson AGX Xavier, launched in 2018, marked a significant advancement with a 512-core Volta GPU, specifically designed for demanding AI tasks in robotics and edge computing [4]. Released in 2019, the Jetson Nano offered a more affordable, compact solution featuring a 128-core Maxwell GPU, and it has rapidly gained popularity among hobbyists and researchers [5, 6]. The Jetson Xavier NX, released in 2020, offered capabilities similar to the AGX Xavier but in a more compact and power-efficient form, featuring a 384-core Volta GPU [7]. Each iteration of the Jetson series has been designed to meet specific needs in the embedded systems market, from powerful, energy-efficient computing to advanced AI applications.

This book highlights Jetson Nano embedded system. The primary reason for selecting this device is its widespread adoption in academic courses and research projects. This embedded systems is frequently utilized in undergraduate class projects and laboratory experiments as it can execute intelligent tasks, such as computer vision-based hardware control, motor operations, data collection, and measurement using sensors. Currently, many companies utilize these Jetson systems to develop their research and commercial AI products. For example, the spot robot from Boston Dynamics uses the Jetson Xavier NX module for data processing and vision-based inspections [8]; ANYmal C-legged robot from ANYbotics can utilize the Jetson AGX Xavier extension for deep learning and computer vision tasks [9]; Skydio inspection drone uses Jetson TX2 for autonomous inspections [10]; and Neurala's Vision AI inspection and automation system employs various Jetson systems to enhance production scalability, reduce waste, and adapt to workforce changes, among other applications [11]. The use of Jetson Nano is also extensive despite its GPU configuration (128-core Maxwell architecture) being relatively low compared to Jetson TX2 (256-core Pascal architecture), Xavier NX (384-core Volta architecture), AGX Xavier (512-core Volta architecture),

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

and Jetson Orin Nano (1024-core Ampere architecture) and other Orin NX/AGX series. Aetina's Internet of Things (IoT) products employ the Jetson Nano for various AI applications [12], MyCobot 280, a collaborative robot arm, uses the Jetson Nano for vision-based intelligent controls [13], and numerous other industrial and academic researchers also utilize the Jetson Nano extensively. Each version of the Jetson series has been developed to address specific needs in the embedded systems market, ranging from powerful, energy-efficient computing to cutting-edge AI at the edge. This has established NVIDIA's Jetson series as one of the major go-to platforms for developers incorporating AI into embedded systems.

5.1.1 Jetson Nano

In this book, we used the Jetson Nano by Nvidia to showcase AI capabilities and programs on embedded devices. It is equipped with a quad-core ARM Cortex-A57 CPU that balances performance with power efficiency and a 128-core Maxwell GPU for handling complex visual computing tasks, making it perfect for image and video processing. It also includes 4 GB 64-bit LPDDR4 RAM, enabling effective multitasking across various applications [14]. The device is also designed for energy efficiency, allowing continuous operation without excessive power consumption.

The Jetson Nano is available in two main versions: one with 2 GB RAM and the other with 4 GB RAM. Although both versions share similar CPU and GPU specifications, the difference in RAM can impact their overall performance and suitability for various tasks. Selecting between the 2 GB and 4 GB versions should be based on specific project requirements, including dataset size, task complexity, and budget constraints. In this book, we have used the 4 GB version to demonstrate programming examples. Additionally, according to NVIDIA, the Jetson Nano 2 GB Developer Kit has reached its end of life and is no longer available for purchase.

A picture of the Jetson Nano 4 GB with different module names is given in Figure 5.1.

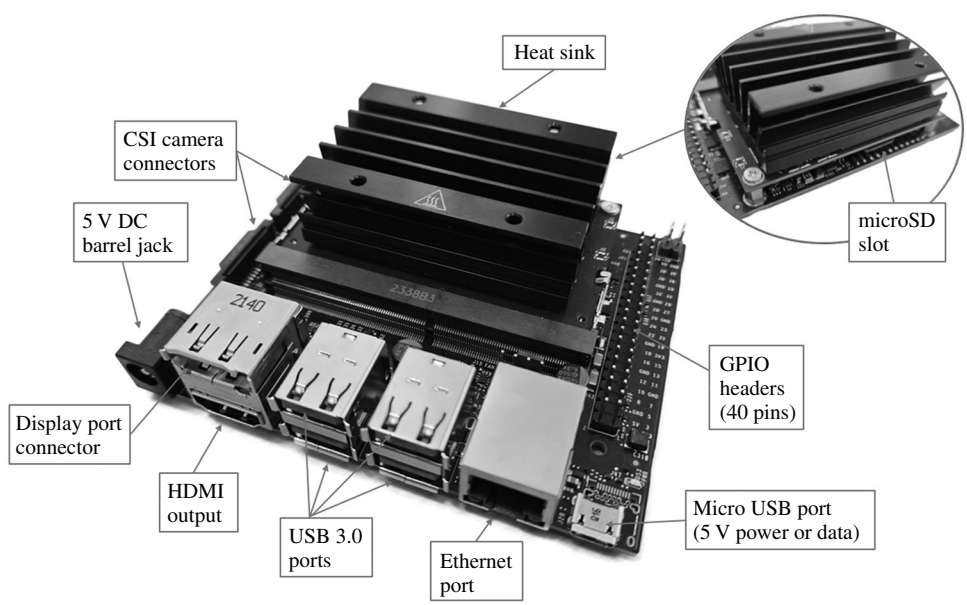


Figure 5.1 Jetson Nano 4 GB board and its modules.

5.1.2 Hardware and Power Requirements

The Jetson Nano can be powered using a 5 V, 4 A power barrel jack, which is recommended for reliable operation, especially when using peripherals. Alternatively, a 5 V, 2 A power source can be used via the micro universal serial bus (micro-USB) port, but this is not recommended for high-performance modes or heavy peripheral use. The high-definition multimedia interface (HDMI) port supports HDMI 2.0 with minimal impact on power consumption, while the DisplayPort supports DisplayPort 1.2 with similar efficiency. The Gigabit Ethernet port provides high-speed connectivity, and two mobile industry processor interface (MIPI) camera serial interface (CSI)-2 camera connectors can be used for compatible camera use. The universal serial bus (USB) 3.0 ports consist of four Type-A ports, each capable of providing up to 1 A at 5 V, with total power usage depending on connected devices. All four USB ports are connected to the Jetson Nano module via a USB 3.0 hub built into the carrier board. The general-purpose input/output (GPIO) header of the Jetson Nano has a 40-pin header for interfacing with external hardware components, and the micro secure digital (microSD) card slot is used for storage and booting the operating system (OS) from a microSD card. The fan header is a 2-pin header for connecting an optional cooling fan. For a detailed overview of various components of the Jetson Nano, it is recommended to refer to the official Jetson Nano documentation available at [15].

The Jetson Nano's power requirements can also vary based on the connected peripherals and the mode in which it operates. It has two modes for power consumption: 5 W and 10 W. In the 5 W mode, the Jetson Nano operates with a power consumption of approximately 5 W, suitable for lower power applications and enabled by default. In the 10 W mode, the Jetson Nano can consume up to 10 W, allowing for higher performance, suitable for applications requiring more computational power. Using a 5 V, 4 A power supply through the barrel jack is generally recommended for most use cases to ensure stable and reliable operation. The hardware components that are used in this book are listed in Table 5.1.

Table 5.1 Descriptions of the hardware components used and alternatives.

Item name	Descriptions	Alternatives
Jetson Nano board	NVIDIA Jetson Nano Developer Kit (V3) (https://www.sparkfun.com/products/16271)	Other well-known online retailers, such as Amazon, AliExpress, Waveshare, etc., also sell the Jetson Nano 4 GB developer kit. Python programs and examples used in this book are also compatible with other versions of Jetson boards
Power supply	5 V, 4 A (4000 mA) switching power supply (https://www.adafruit.com/product/1466) Or (https://www.sparkfun.com/products/15352)	Any other 5 V, 4 A power supply adapter with a "standard" 5.5 mm OD, 2.1 mm ID positive tip connector will work
Keyboard	Logitech – K120 Full-size Wired Membrane Keyboard (https://www.logitech.com/en-us/products/keyboards/k120-usb-standard-computer.920-002478.html)	Any other USB keyboard can be used

(Continued)

Table 5.1 (Continued)

Item name	Descriptions	Alternatives
Mouse	DELL - Optical Mouse MOC5UO USB Wired (https://www.amazon.com/Dell-3-button-Optical-Scroll-XN966/dp/B0029ND59O)	Any other USB mouse can be used
Monitor	Lenovo D22e-20 Monitor (https://www.amazon.com/Lenovo-Customizable-Brightness-Flicker-Free-FreeSync/dp/B09QXMHNZ8?th=1)	Any other HDMI monitor can be used
USB Camera	Logitech C270 HD Webcam—720p (https://www.logitech.com/en-us/products/webcams/c270-hd-webcam.960-000694.html)	Although many other similar webcams can be used, this Logitech C270 is highly recommended
Wi-Fi Adapter	Edimax 2-in-1 Wi-Fi 4 N150 and Bluetooth 4.2 Nano USB Adapter (https://www.sparkfun.com/products/22824)	Any other Wi-Fi adapter can be used, but the installation procedure may vary from the one described in this chapter
HDMI Cable	Raspberry Pi Official HDMI Cable (1 m) (https://www.sparkfun.com/products/17387)	Any high-speed male-to-male HDMI cable that supports 4 K resolution will work
Micro USB to USB Cable	5 ft USB to Micro-USB Cable (https://www.amazon.com/Staging-Product-Not-Retail-Sale/dp/B0741WGQ36)	Any USB to Micro-USB data cable will work
micro secure digital (microSD) Card	256 GB Extreme micro secure digital extended capacity (SDXC) ultra-high speed (UHS)-I Memory Card with Adapter (https://www.amazon.com/dp/B09X7CRKRZ/ref=twister_B0BHJXCFXW?_encoding=UTF8&th=1)	Any other micro secure digital high capacity (SDHC) UHS-I with similar MB/s will work
SD Card Reader	USB 3.0 Portable Card Reader for SD, SDHC, SDXC, MicroSD, MicroSDHC, micro secure digital extreme capacity (MicroSDXC) (https://www.amazon.com/SmartQ-C307-Portable-MicroSDHC-MicroSDXC/dp/B06ZYXR7DL?th=1)	Other High-speed USB 3.0 Interface will work

5.2 Jetpack Installation

JetPack is a comprehensive software development kit (SDK) provided by NVIDIA, specifically designed for their Jetson series embedded computing boards [16]. It has essential tools, libraries, and APIs needed to develop AI computing platforms [17]. JetPack significantly improves Jetson Nano's capabilities for deep learning and machine learning projects, making it great for developers working in robotics, IoT, and other advanced computational fields.

JetPack has a Board Support Package for necessary drivers and Linux for Tegra, which is an Ubuntu-based OS optimized for Tegra processors to ensure enhanced performance and stability. The package also integrates CUDA-X AI, a collection of libraries leveraging NVIDIA compute unified device architecture (CUDA) technology for accelerated processing across various domains, such as AI and deep learning. Additionally, it includes the CUDA Toolkit for developing GPU-accelerated applications, TensorRT, and the CUDA deep neural network library (cuDNN) for efficient deep learning inference and computer vision tasks.

To install the JetPack on a Jetson Nano, we will need a host computer (Windows, macOS, or Linux) to download the SD card image for the SDK. The current version of the Jetson Nano Developer Kit SD card image is available at [18]. If needed, we can access all the previous JetPack versions compatible with the Jetson Nano in the JetPack Archive [19], and all documentation and software can be downloaded from the Jetson Download Center [20].

In this book, we used the pre-built Jetson Nano image from Qengineering's GitHub repository, which offers significant advantages, particularly in terms of time savings and ease of use. We highly recommend downloading the image file from https://github.com/Qengineering/Jetson-Nano-image and flashing it on a microSD card [21]. The image comes with a pre-configured environment that includes many essential libraries, drivers, and tools for machine learning and deep learning applications. They are all optimized for the Jetson Nano's hardware. This will allow users to begin working on projects immediately without the need to install and configure packages, such as open source computer vision library (OpenCV) and PyTorch. The repository is widely recognized, providing access to community support and frequent updates, which include bug fixes, new features, and compatibility improvements. The image file can also be downloaded from the Google Drive link https://drive.google.com/file/d/1eWlLx1mHVLkwaK2zMNNZI-rnvNEmBbk7/view?usp=sharing.

After downloading the image file (JetsonNano.img.xz), the next step is to format the microSD card. It is recommended to use a microSD card with at least 128 GB (e.g. 128 GB Extreme microSDXC) of storage space. In this book example, we used a 256 GB microSD card because we needed to download multiple packages and modules for different chapters. To proceed with formatting, connect the microSD card to a host computer via an SD card reader. Download and install the "SD Memory Card Formatter" for Windows by the SD Association [22]. Then launch the SD Card Formatter, select the correct drive (verify disk capacity), and format it as shown in Figure 5.2. The formatted SD card can then be used to flash the "JetsonNano.img.xz" file in the host computer using a USB port via the SD card reader.

Next, download and install the balena Etcher application from https://etcher.balena.io/#download-etcher or another flash tool compatible with the host computer's operating system. Use the Etcher software to flash the downloaded image (JetsonNano.img.xz) into the microSD card, as illustrated in Figure 5.3.

Before flashing the microSD, "balenaEtcher" may show a warning that you are about to erase an unusually large drive. Press "Yes, I'm sure" to confirm that. During the Etcher verification process, multiple file explorer windows and prompts to format the disk may appear. These occurrences can vary based on the operating system and firewall settings. You might also see a prompt stating that the "drive is not accessible." Close all these explorer windows and prompts, and allow the verification process to complete in the Etcher software.

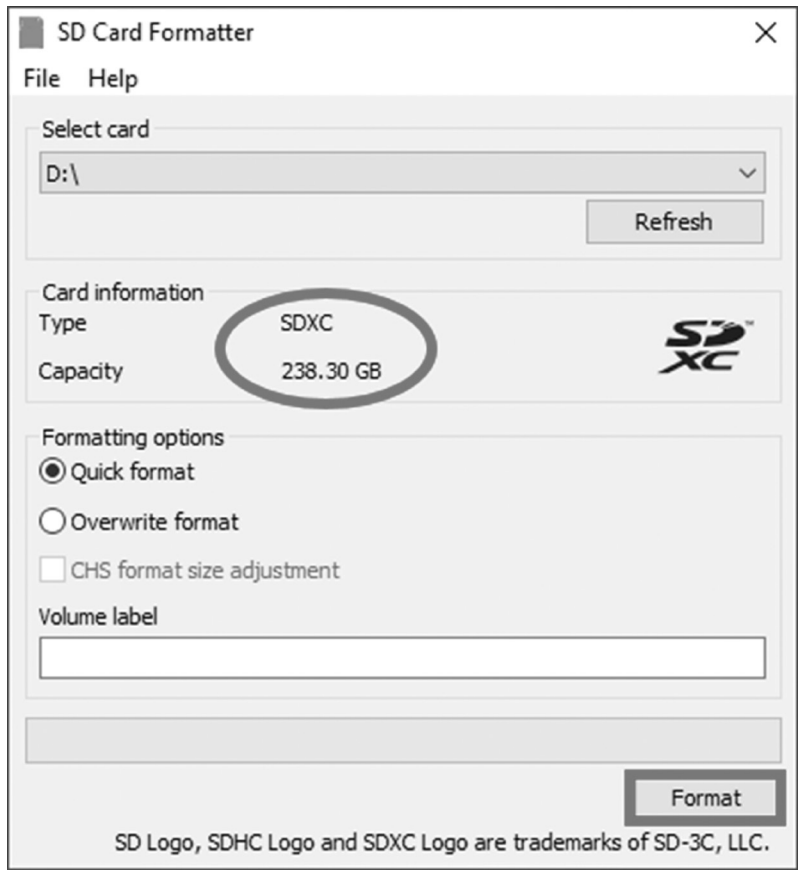


Figure 5.2 Check the memory capacity of the microSD card and use the SD Memory Card Formatter to format it.

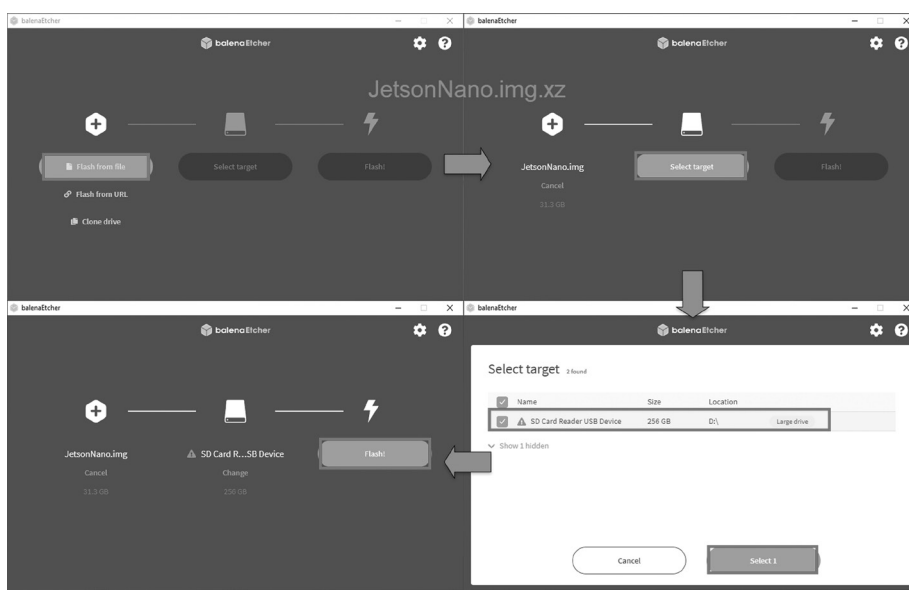


Figure 5.3 Choose the "JetsonNano.img.xz" file, select the target, and then press the flash button to install the operating system image on Jetson Nano.

5.3 Direct Setup

For the initial setup, an Ethernet cable can be used to download the Wi-Fi driver for the Jetson Nano. At this stage, all peripheral connections to the Jetson Board should match those shown in Figure 5.4. To use the 5 V, 4 A power supply and disable the micro USB power supply, connect the DC enable jumper near D63. When power is supplied to the device, the power light emitting diode (LED) will light up green, as illustrated in Figure 5.4.

After connecting the peripheral devices to the Jetson Nano, insert the newly flashed microSD card into the microSD slot (Figure 5.1) for the first run. Then, power the Jetson Nano board using the 5 V, 4 A power supply with all the peripherals connected, as shown in Figure 5.4.

If we had chosen to use the official JetPack image, during the first bootup, the system would go through some basic configuration setup, including username and password. But this prebuilt image file (JetsonNano.img.xz) already has the configured username and password (username: jetson & password: jetson).

5.3.1 Increase Root Partition Size

After the first boot, we will need to increase the root partition size to install heavy applications and use more space. With PyTorch, OpenCV, and other necessary libraries installed in the pre-built image, only about 6.8 GB of space will remain available in the root. We may use the "df -h" command from the terminal to check the file system's total size, used space, available space, and the percentage of space used, along with the mount points (Figure 5.5).

Then, we should update the system packages using the "sudo apt-get update" command. This command refreshes the metadata about available packages, including their versions, dependencies, and other relevant details. To upgrade all currently installed packages to their latest versions,

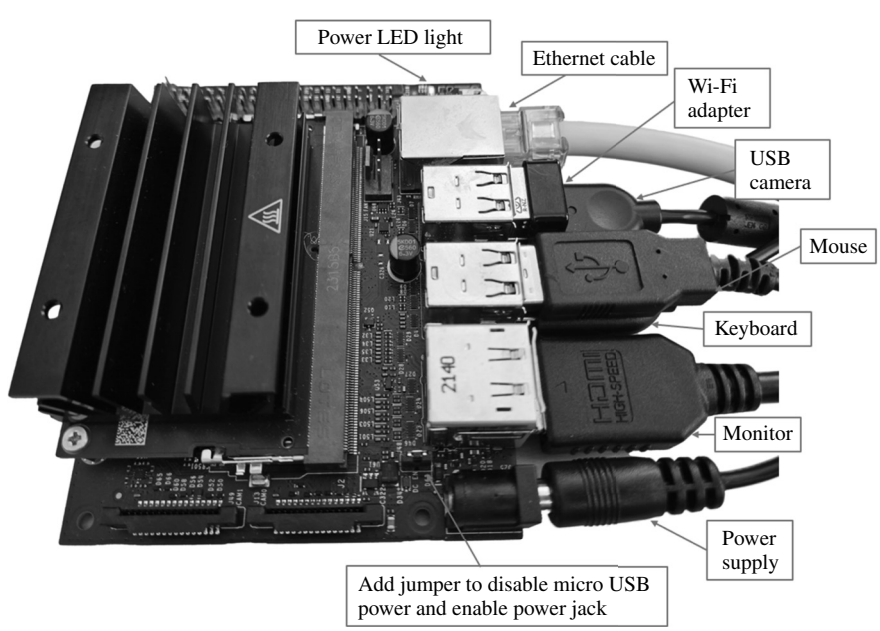


Figure 5.4 Peripheral connections to the Jetson board for using the power jack for 5 V, 4 A supply and an Ethernet cable for internet access.

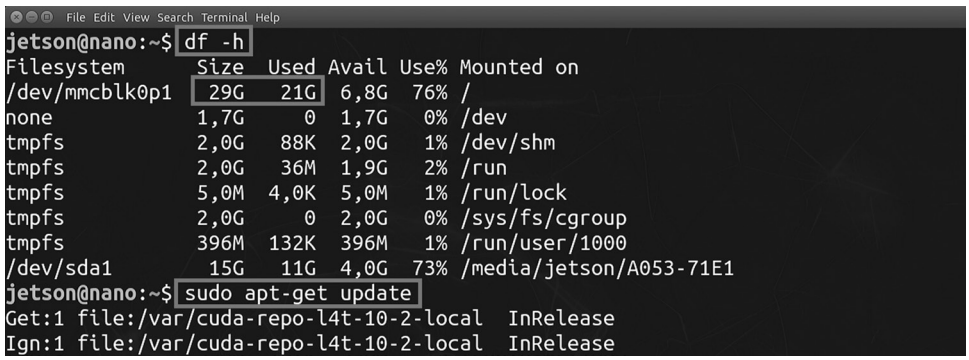


Figure 5.5 Check the root file system's size and available spaces and update packages.

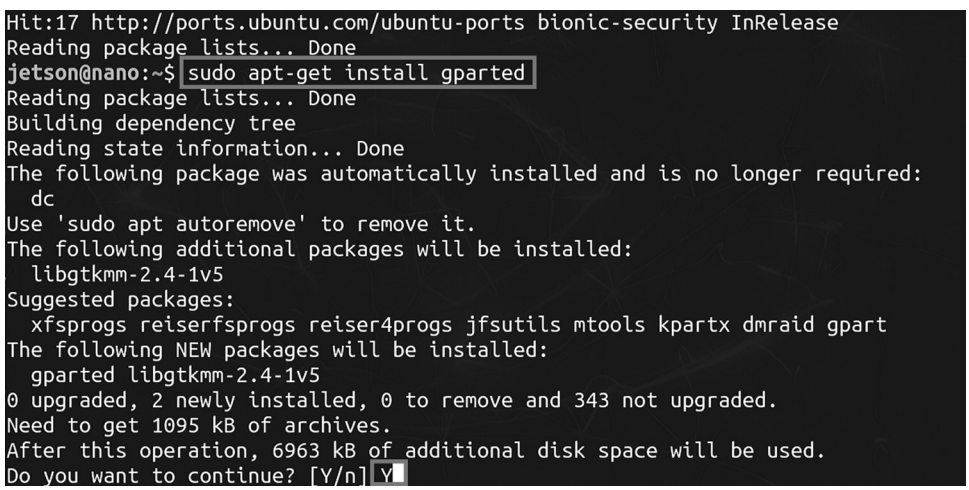


Figure 5.6 Install the partition editor "gparted" to increase the size of the root file system.

use the "sudo apt-get upgrade" command. This step is crucial for effectively using docker files on the Jetson Nano. During the upgrade, we will be prompted to choose whether to automatically restart the docker daemon. Select "yes" and press enter. Some configuration files have changed between docker versions, and this upgrade process will help apply those updates.

Next, we'll install a graphical partition editor called "gparted" on the system. This tool can be used for various tasks such as creating, resizing, moving, and deleting partitions on storage devices. Since we opted for a 256 GB microSD card, there's plenty of space available for expansion. Even if we're using a smaller SD card, such as 128 GB or 64 GB, "gparted" can help allocate additional space to the root file system. To install "gparted," run the "sudo apt-get install gparted" in the terminal, press Y, and hit enter to proceed with the installation (Figure 5.6).

After installing gparted, launch this application (using "sudo gparted") with superuser (root) privileges to manage the disk partition. This command will open a graphical interface where we can manage the partitions visually. Figure 5.7 shows that we have about 209 GB of unallocated space in the 256 GB microSD storage.

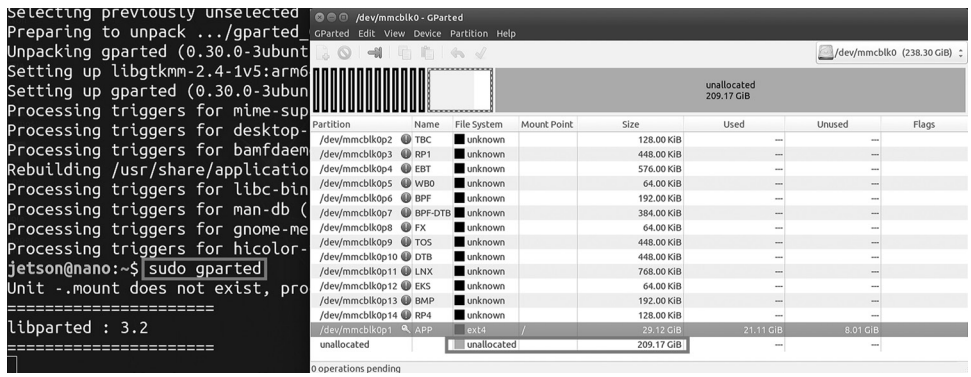


Figure 5.7 Launch "gparted" with superuser privileges to manage the root partition.

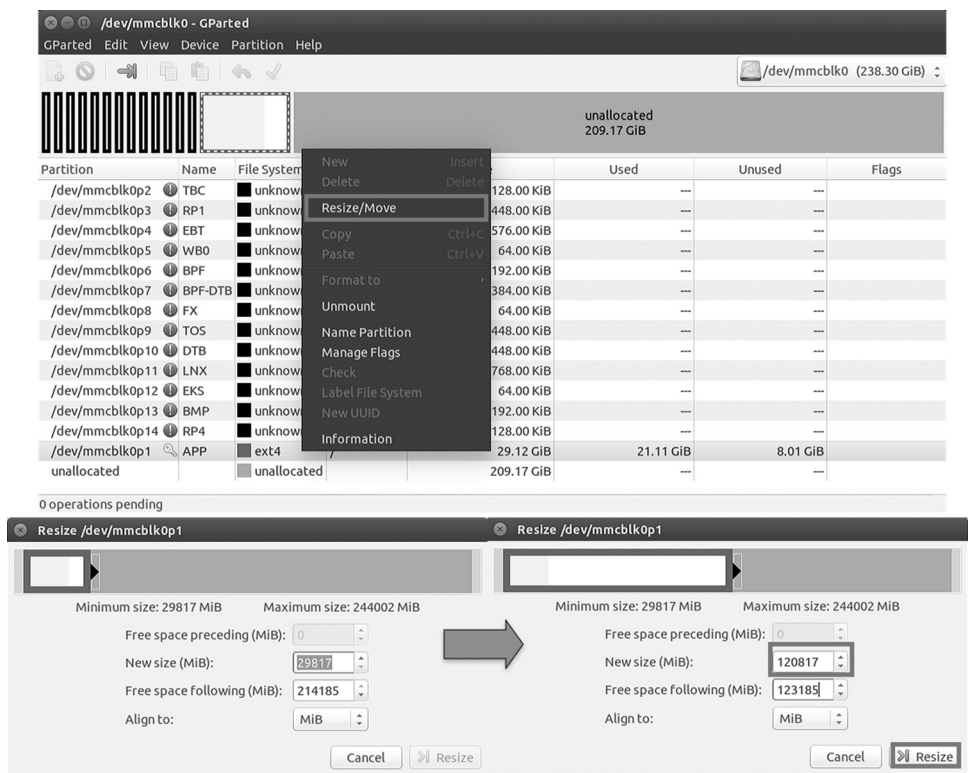


Figure 5.8 "Resize" root partition size from 29 GB to 120 GB using "gparted."

Right-click on the partition labeled "/dev/mmcblk0p1" with the "ext4" file system, and choose the "Resize/Move" option, as shown in Figure 5.8. Then change the current size of the partition from 29 GB (29817) to 120 GB (120817). We can choose any other suitable size based on the disk space. After changing the values in new size, free space on the disk will be updated automatically. Press "Resize" to reallocate the memory.

Select the green checkmark available at the top of the "gparted" application. A prompt window will appear to warn that this operation may cause loss of data. Select "Apply" to proceed (Figure 5.9).

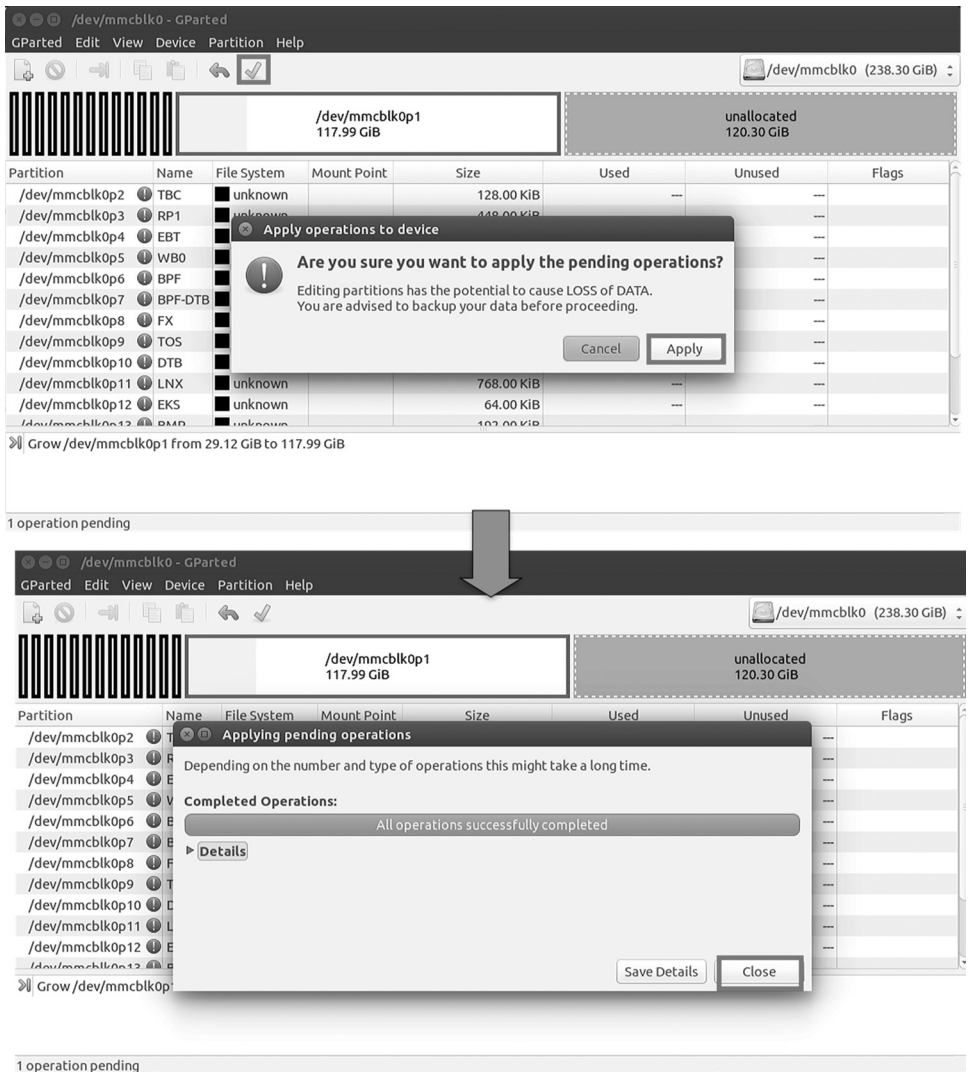


Figure 5.9 Apply new partition size using "gparted."

Once the partition process is complete, close the "gparted" window and restart the system by running "sudo reboot now". After the reboot, you can confirm the new partition size by using the "df -h" command in the terminal. As shown in Figure 5.10, 91 GB of space is now available in the root system. This additional space will allow us to build images with docker engines and install additional deep learning libraries (if needed) from the root system.

5.3.2 Other Settings

We should change the time, date, and location manually from the desktop settings > time & data option, as they can interfere with some applications. Then, change the password from the Jetson's terminal using the command "sudo passwd jetson" (Figure 5.11). We will be prompted to enter and

```
Unit -.mount does not exist, proceeding anyway.
=======================
libparted : 3.2
=======================
jetson@nano:~$ sudo reboot now
```

```
jetson@nano:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/mmcblk0p1  117G   21G   91G  19% /
none            1,7G     0  1,7G   0% /dev
tmpfs           2,0G   88K  2,0G   1% /dev/shm
tmpfs           2,0G   28M  2,0G   2% /run
tmpfs           5,0M  4,0K  5,0M   1% /run/lock
tmpfs           2,0G     0  2,0G   0% /sys/fs/cgroup
tmpfs           396M  120K  396M   1% /run/user/1000
/dev/sda1        15G   11G  4,0G  73% /media/jetson/A053-71E1
jetson@nano:~$
```

Figure 5.10 Reboot the system and verify available space in the root file system.

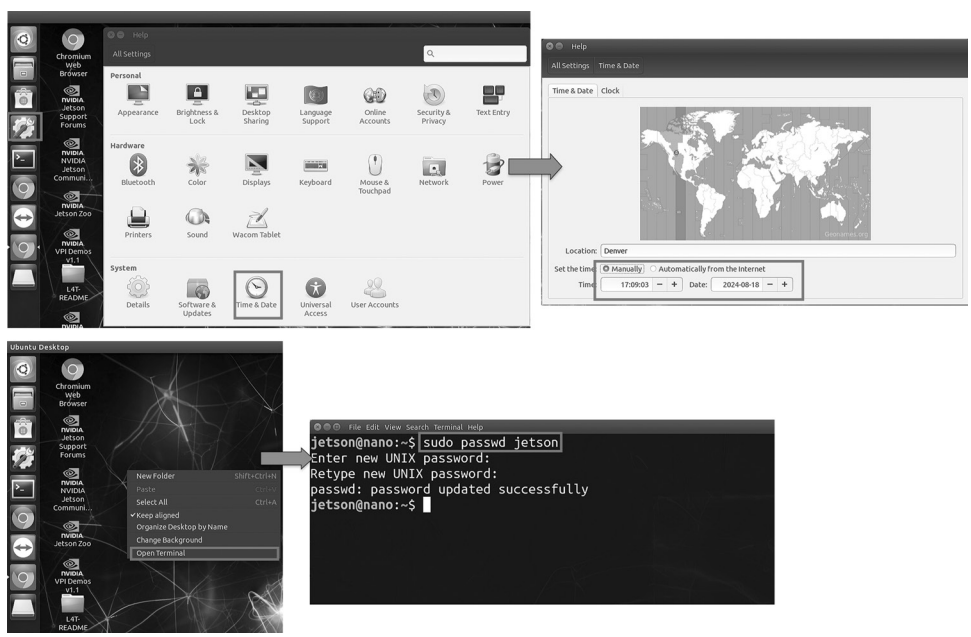


Figure 5.11 Manually update the location, date, and time, and change the current password.

confirm the new password. For convenience, we chose a simple one-word password, as it will need to be entered frequently during package installations. You may change the password later if needed, and it is always advised not to use an easy or single-word password for security reasons. Also, note that the browser login keyring will be the same (jetson).

Next, open the Linux command terminal by double-clicking the terminal icon or right-clicking and selecting the "Open Terminal" option. In the terminal window, update and upgrade the system

by entering "sudo apt-get update" and "sudo apt-get upgrade." The update command updates the local package index on the Jetson Nano and fetches the latest package lists from the repositories configured on the system. The upgrade command, on the other hand, installs the latest versions of all the packages currently installed on the Jetson Nano. If prompted to select "y/n" to download package files, select "y" to proceed. The upgrade step might need some time (10–20 minutes) to complete. During the upgrade operation, it will also ask about the settings to restart the docker daemon automatically. Respond (enter command) with "Yes" and select "Next" to proceed. Once the upgrades are completed, restart the Jetson by entering the command "sudo reboot now."

5.3.3 Wi-Fi Driver

The Wi-Fi adapter we used is called "Edimax 2-in-1 Wi-Fi 4 N150 & Bluetooth 4.2." The Linux Ubuntu driver for this adapter can be downloaded from [23]. The driver zip file name is "EW-7611ULB V2 Linux Wi-Fi & Bluetooth Driver Support Kernel: Fedora : 4.5~5.11, ubuntu : 4.4~5.11." Download and unzip the driver file, as shown in Figure 5.12.

To install the driver, navigate to the extracted folder's "WIFI" directory and open the terminal by right-clicking and selecting "Open Terminal." Alternatively, you can manually access the "WIFI" folder by using the cd "folder name" command from the desktop. In the terminal, enter the following three commands sequentially to compile and install the driver from the source code.

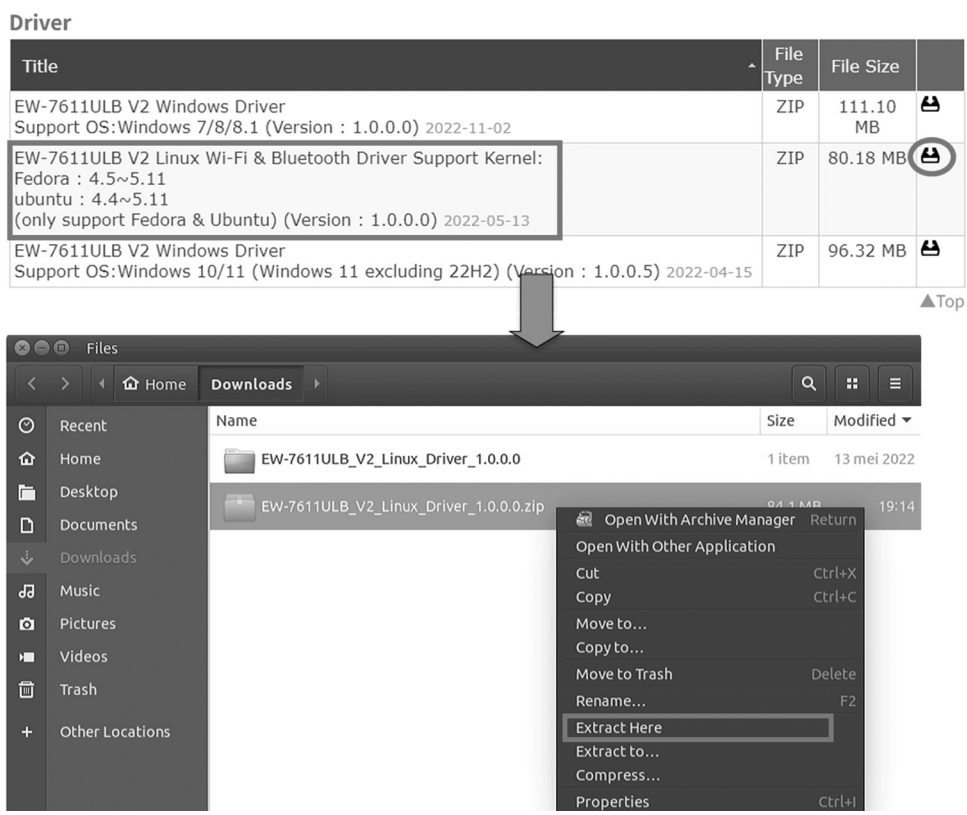


Figure 5.12 Download the driver of the Wi-Fi adapter "Edimax 2-in-1 Wi-Fi 4 N150 & Bluetooth 4.2" and extract the zip file in the download folder.



Figure 5.13 Steps for compiling and installing Wi-Fi and Bluetooth driver for EW-7611ULB Edimax.

$ export ARCH=arm64
$ make
$ sudo make install

The installation procedure shown in this section is available in a document on Sparkfun's seller website (https://www.sparkfun.com/products/22824). After installing the Wi-Fi driver, navigate to the driver's BT/Linux folder in the terminal and execute the following command, then reboot the system. These steps are shown in Figure 5.13.

$ sudo make install INTERFACE=usb
$ sudo reboot now

After rebooting, the Wi-Fi icon should appear in the top right corner, and you should be able to connect to a network using the password. If you are using an "eduroam" network, the Jetson's Wi-Fi connection may not work properly due to various firewalls and administrative access restrictions at different educational institutions. In such cases, it's better to use a direct Ethernet cable to connect to the internet. If Wi-Fi is required for a mobile robot or project, an Ethernet cable with a router can be used to create a separate Wi-Fi network with its username and password and then connect to that Wi-Fi network. Alternatively, users can use a cell phone network to create Wi-Fi hotspots. For home Wi-Fi, if the user has admin access, the connection should be smooth.

Using a Wi-Fi network for internet access is not mandatory for following the examples or case studies given in this book. We can always use an Ethernet cable to connect the Jetson to download or update packages, software, docker files, and libraries when a Wi-Fi connection is not available. Then, we can run programs on the Jetson without an internet connection for regular use.

5.4 Configure Visual Studio Code on Jetson

Visual Studio Code (VS Code), developed by Microsoft, is a lightweight and powerful source code editor that supports various programming languages. It offers built-in features, such as debugging, task running, and version control. Its high customizability allows users to install extensions, making it suitable for diverse development tasks. VS Code is an ideal choice for Jetson Nano due to its efficiency, remote development capabilities, and extensive extensions for languages such as Python, JavaScript, C#, C++, etc. Its robust debugging tools are crucial for AI and machine learning projects, and its cross-platform compatibility ensures a consistent development environment for multiple users. Additionally, the large and active VS Code community provides extensive support and resources, which can be particularly helpful for developers working on Jetson Nano.

In this section, we will install the VS Code IDE, configured specifically for the Jetson Nano. First, ensure all packages are up to date by entering "sudo apt-get update" in the terminal. This step is crucial before installing new software or performing system upgrades, as it provides the latest information about available packages. Next, use the "sudo apt-get upgrade" command to install packages on the Jetson system to the latest versions available in the repositories. If you have recently updated and upgraded the system, these commands will run but won't make any changes.

After completing the updates and upgrades, navigate to the "Downloads" folder in the terminal and install curl by using the commands "cd Downloads" and "sudo apt-get install curl" as shown in Figure 5.14.

$ cd Downloads
$ sudo apt-get install curl

"curl" is a command-line tool used to transfer data with URLs and is commonly used for downloading files, making HTTP requests, and interacting with APIs. It is widely used in development, testing, and automation for various data transfer and interaction tasks. Next, use the following command to download Jetson Nano compatible VS code (Figure 5.15):

$ curl -L https://github.com/toolboc/vscode/releases/download/1.32.3/code-oss_1.32.3-arm64.deb -o code-oss_1.32.3-arm64.deb

To install the downloaded code-oss Debian package, use the command in the terminal, as shown in Figure 5.16. After the installation process, enter code-oss to run the VS code for the first time.

$ sudo dpkg -i code-oss_1.32.3-arm64.deb
$ code-oss

In VS Code, navigate to the extension section and search for the Python extension to install it. Then, use the shortcut "ctrl+alt+p" to open the interpreter selection interface in VS Code and choose the Python 3 version, as shown in Figure 5.17.

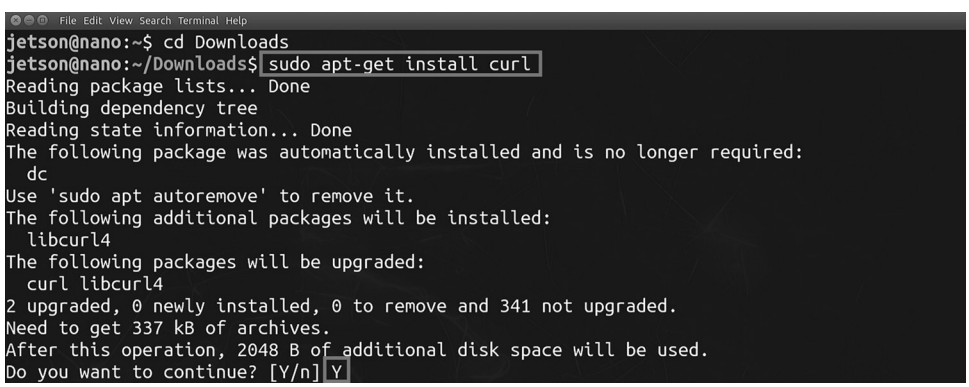


Figure 5.14 Install the command-line tool "curl" in Jetson Nano.

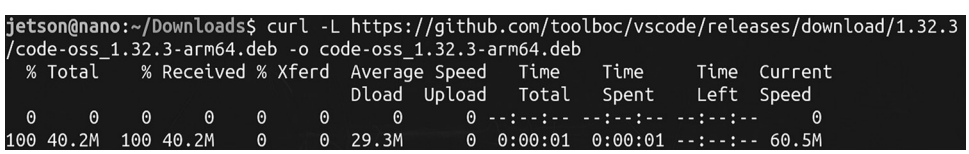


Figure 5.15 Download Jetson Nano compatible VS code package using "curl" location.

```
jetson@nano:~/Downloads$ sudo dpkg -i code-oss_1.32.3-arm64.deb
Selecting previously unselected package code-oss.
(Reading database ... 173362 files and directories currently installed.)
Preparing to unpack code-oss_1.32.3-arm64.deb ...
Unpacking code-oss (1.32.3-1560357964) ...
Setting up code-oss (1.32.3-1560357964) ...
Processing triggers for gnome-menus (3.13.3-11ubuntu1.1) ...
Processing triggers for desktop-file-utils (0.23-1ubuntu3.18.04.2) ...
Processing triggers for bamfdaemon (0.5.3+18.04.20180207.2-0ubuntu1) ...
Rebuilding /usr/share/applications/bamf-2.index...
Processing triggers for mime-support (3.60ubuntu1) ...
jetson@nano:~/Downloads$ code-oss
```

Figure 5.16 Install the code-oss Debian package for Jetson Nano and launch the VS IDE.

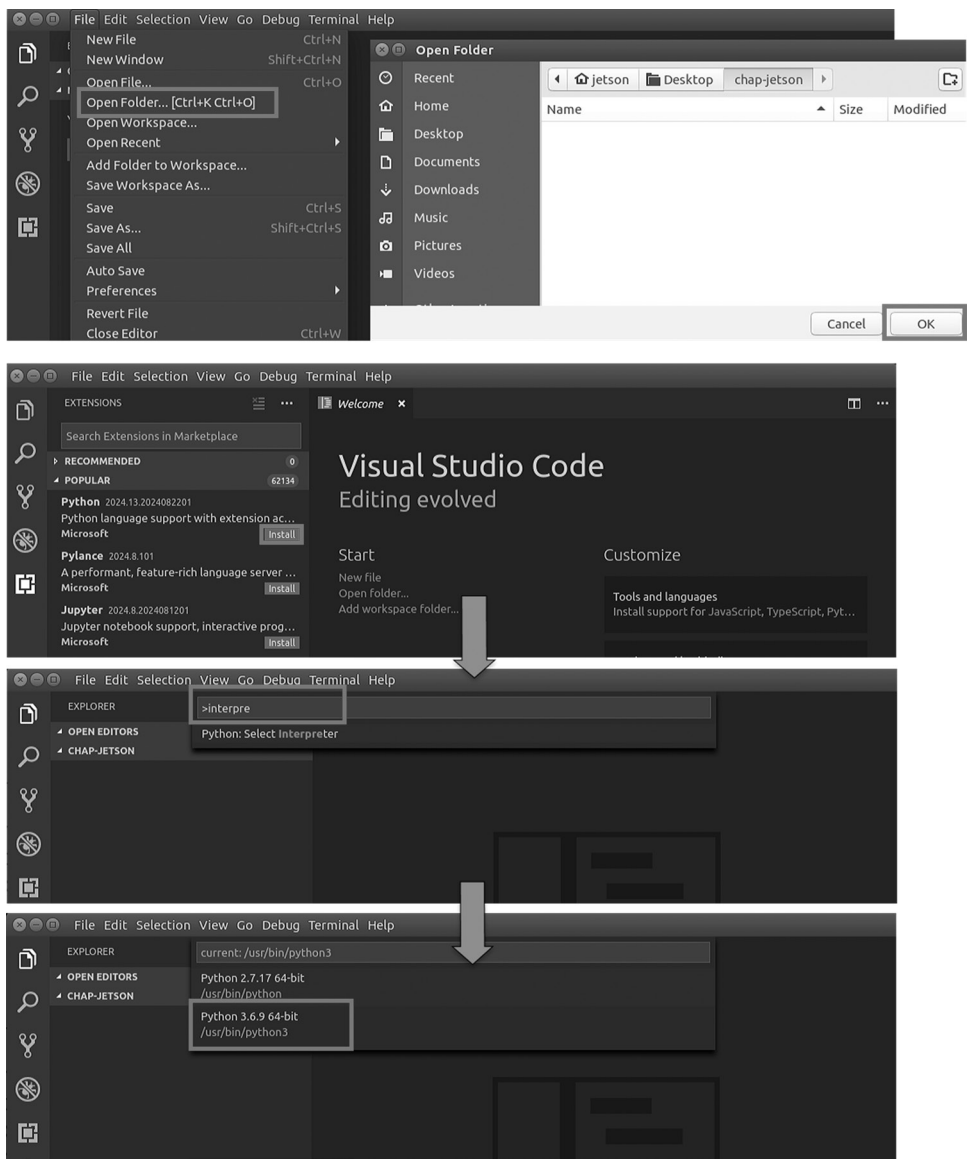


Figure 5.17 Steps for installing Python from extension and selecting interpreter in VS code.

To test a Python program in VS Code, create a new folder named “chap-jetson” (or any other name of your choice) on the Jetson Desktop. Open this folder in VS Code by selecting it from the File menu. Next, create a new file called “example1.py” by clicking on the new file icon. Then, open a new terminal via the terminal menu in VS Code. At this stage, we made some adjustments to the display and layout settings in VS Code. These are optional, and you can either leave them as they are or customize the layout to your preference. We changed the editor window’s font size using “File > Preferences > Settings” and switched the color theme to Light (Visual Studio) via “File > Preferences > Color Theme.” These modifications are shown in Figure 5.18.

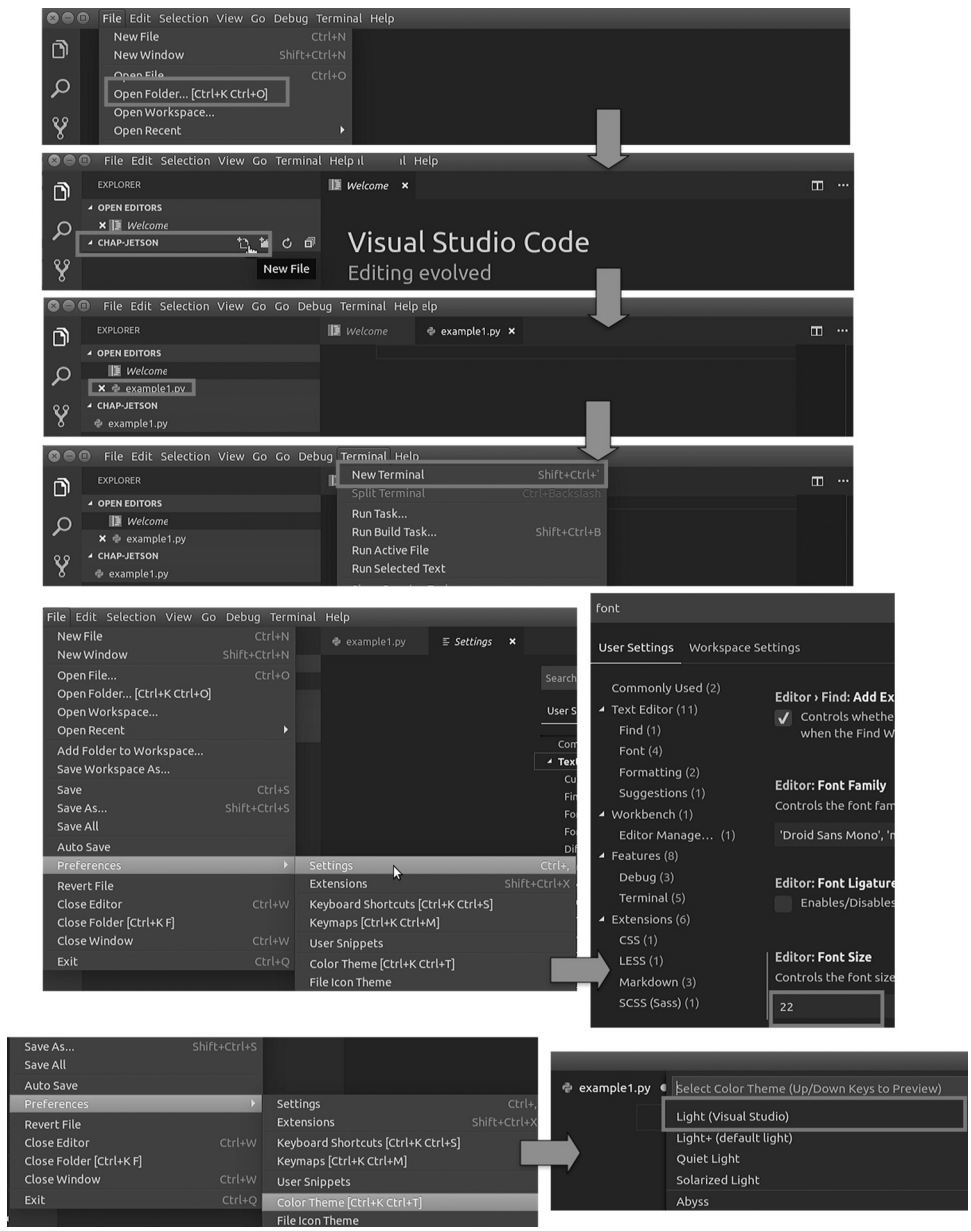


Figure 5.18 Create a new Python file in the specified folder, change the font size, and color theme.

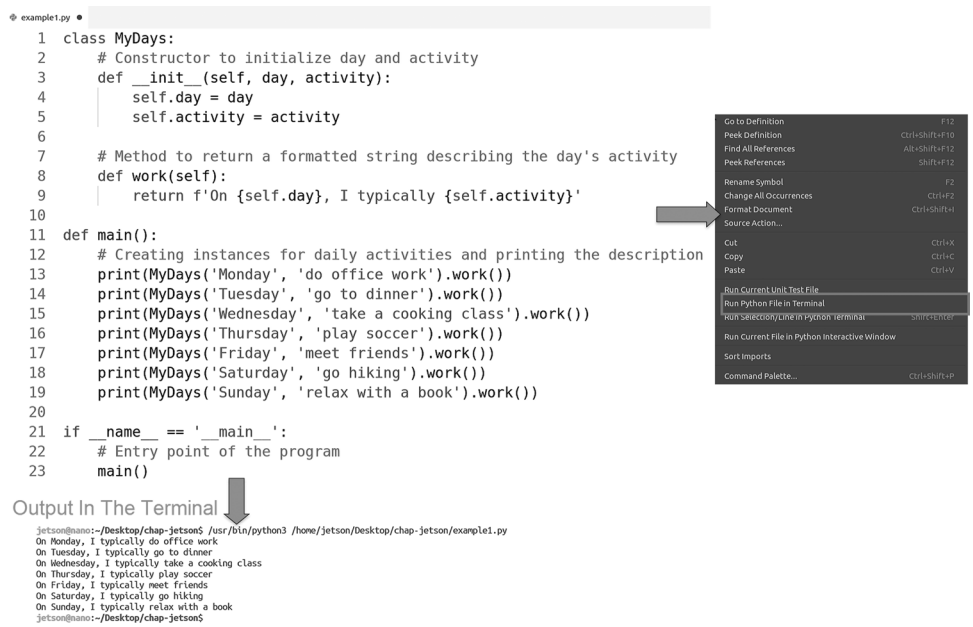


Figure 5.19 Testing "example1.py" Python programming in VS code.

Next, write a simple Python program to test the output, as shown in Figure 5.19. The "example1.py" uses a class called "myDays" to define weekdays and corresponding daily activities, and this is utilized from a main function. To run the program, right-click and select the "run Python File in Terminal" option. The VS code terminal will display the outputs.

5.5 OpenCV and PyTorch in Jetson

The image file we used in Section 5.2 comes with several important libraries and packages pre-installed, including OpenCV and PyTorch. OpenCV is one of the important libraries that we will frequently use, and installing it on the Jetson Nano presents several challenges. The main issue is that the pre-installed version of OpenCV in JetPack lacks CUDA support, which is essential for leveraging the Jetson Nano's GPU capabilities. To get a CUDA-enabled version, we will need to compile OpenCV from source, which requires significantly more RAM and swap space than the Nano typically provides. This process has many steps, and that's why we utilized the image provided by the Q-Engineering [21]. Similar to OpenCV, installing PyTorch also has compatibility issues, as newer PyTorch versions require CUDA 11 and Python 3.7, which are not supported on the Nano's standard JetPack. Also, building PyTorch from source is resource-intensive, requiring over 8 hours on an overclocked Nano and more RAM through swap space. Therefore, using pre-installed OpenCV and PyTorch images from a third party is very convenient. These images not only include major installations such as OpenCV and PyTorch but also come with other essential pre-built packages, such as "matplotlib," "numpy," "torchvision," etc. This setup also ensures compatibility between packages and makes it easier to start working with these tools right away.

5.6 Setting up Jetson Inference

The Jetson inference library is an essential deep learning library that is designed specifically for NVIDIA Jetson platforms, such as the Jetson Nano [24]. It provides a collection of pre-trained models and tools for deploying deep learning-based computer vision applications. Installing the Jetson inference library on the Jetson Nano is very useful as it provides optimized implementations of many popular deep learning models by leveraging the CUDA cores for accelerated performance. By using Jetson inference, developers can easily deploy pre-trained models or train custom ones, leveraging Nano's AI capabilities without having to deal with low-level GPU optimizations. The commands needed to be used for this setup are given below:

$ sudo apt-get update
$ sudo apt-get install git cmake libpython3-dev
$ cd Downloads
$ git clone --recursive --depth=1 https://github.com/dusty-nv/jetson-inference
$ cd jetson-inference
$ mkdir build
$ cd build
$ cmake ../
$ make -j$(nproc)
$ sudo make install
$ sudo ldconfig

First of all, we will need to update Jetson's Jetpack using the command "sudo apt-get update" from the terminal. This will fetch the latest package from the repositories and resolve dependencies based on the most current information. Before any major installations in Jetpack, it is important to update other packages, as it ensures access to the latest versions of software packages and dependencies. After this, use "sudo apt-get install git cmake libpython3-dev" to install "git," "cmake," and "libpython3-dev." These are essential tools and libraries that we need to set up Jetson inference. The "git" is a version control system that allows developers to manage and track changes in their code to support collaborations and download source codes from GitHub. "cmake" is a cross-platform build system that simplifies the process of compiling and building complex software projects across different platforms. And the "libpython3-dev" is a package that has the header files and libraries for building Python extensions and embedding Python in C/C++ projects. This installation process is shown in Figure 5.20.

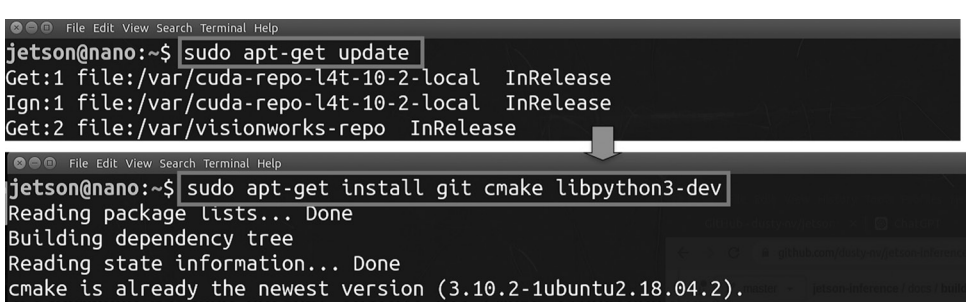


Figure 5.20 Update Jetpack and install "git," "cmake," and "libpython3-dev."

Next go to the "Downloads" directory (by using "cd Downloads") and use the command "git clone --recursive --depth=1 https://github.com/dusty-nv/jetson-inference" to copy "jetson-inference" repository in the Jetson. By using the "--depth=1" option, it performs a shallow clone, i.e. downloads only the latest commit to reduce the amount of data copied. Additionally, the "—recursive" option ensures that any submodules within the repository are also cloned and all dependencies are included (Figure 5.21). Then enter the "jetson-inference" directory (by using "cd jetson-inference"), and to make the build files clean and separate, create another folder in it named "build" (mkdir build). Access the "build" folder (using "cd build") and generate the build files in it using "cmake ../" command. During the build process, an installer prompt will ask to install "PyTorch" library. Choose "skip" to avoid the installation, as this version of PyTorch and its dependencies could conflict with the existing installations of PyTorch, OpenCV, and Numpy, and could potentially make these packages unusable. These steps are illustrated in Figure 5.21.

To compile the codes using "make" build system and to use available threads, use the command "make -j$(nproc)." This will dynamically retrieve the number of cores and will speed up the compilation process. After compilation, use "sudo make install" to install compiled binaries and headers. The "sudo ldconfig" updates Jetpack's dynamic link to ensure that the system can link to dynamic libraries correctly while running Jetson inference. The "git submodule update –init" initializes and updates any submodules from the "jetson-inference" repository (Figure 5.22).

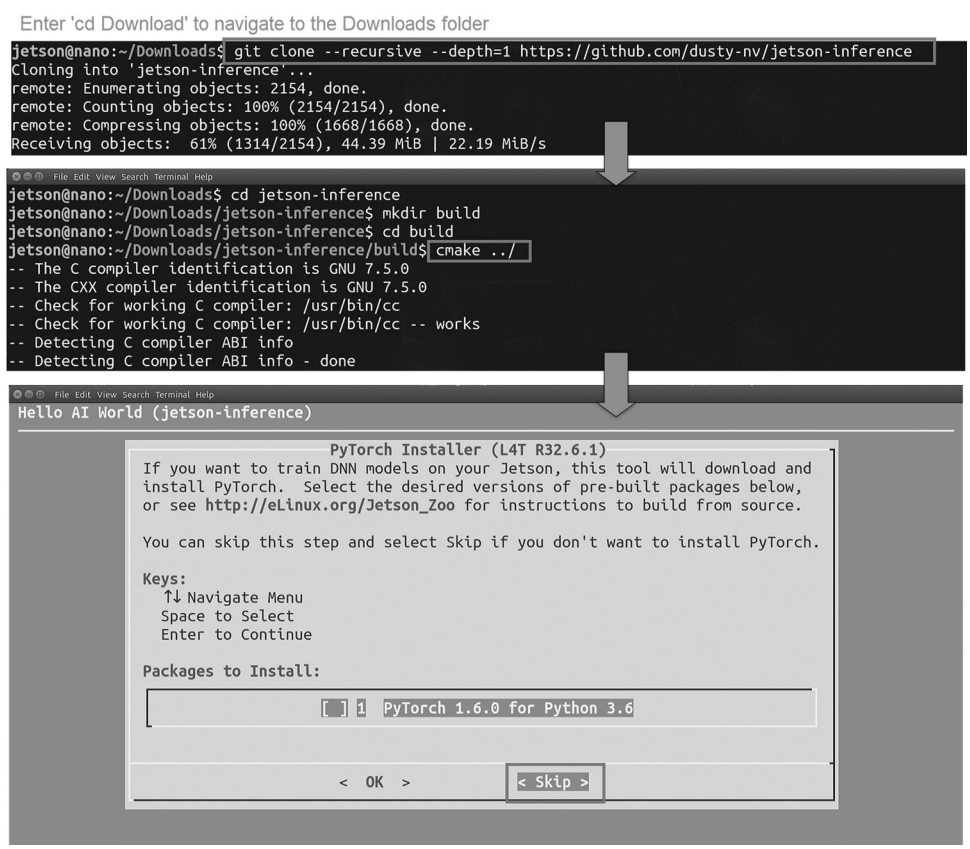


Figure 5.21 Clone the "jetson-inference" repository and build files inside the "build" folder using "cmake ../."

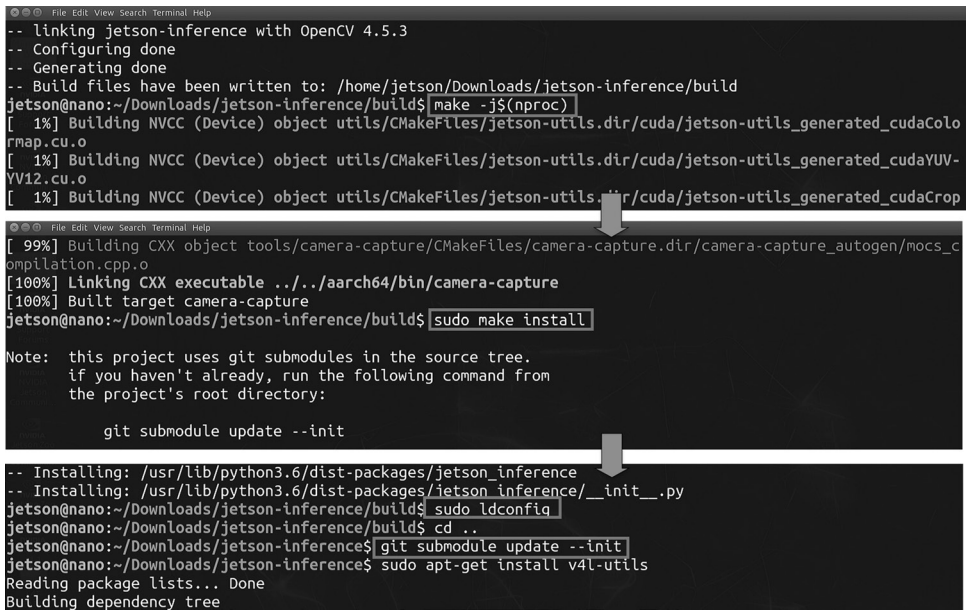


Figure 5.22 Compile codes using "make" build, install compiled binaries, update dynamic links, and initialize and update submodules.

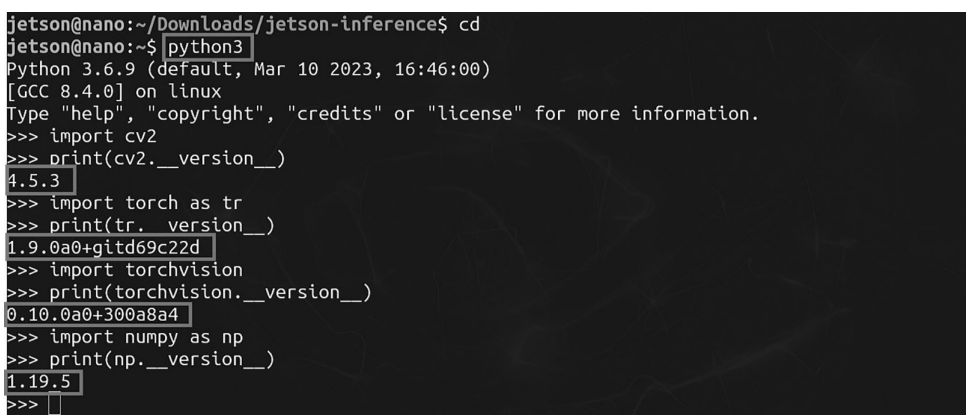


Figure 5.23 Start Python from the terminal and check the installed versions of "cv2," "torch," "torchvision," and "numpy."

At this point, most of the essential libraries have been installed on the Jetson Nano. To verify they are functioning correctly, we can start Python by typing "python3" and then check the installed versions of "cv2," "torch," "torchvision," and "numpy," as shown in Figure 5.23. It is important to note that when using OpenCV in Python, it should be imported before any other libraries, particularly before PyTorch. Failing to do so may lead to runtime errors and conflicts between the two libraries due to differences in their dependencies, particularly with CUDA and cuDNN.

After confirming these libraries, restart the system from the terminal using the command "sudo reboot now."

5.7 OpenCV Library and Test Video Capture Functionality

One of the most important libraries for computer vision applications is OpenCV. It is an open-source library designed for computer vision and machine learning tasks [25]. It has a wide variety of tools for analyzing visual data and supporting activities such as image and video manipulation, object detection, and various machine learning applications. Key features of OpenCV include functions for image manipulations, such as filtering, transformation, resizing, cropping, and rotating. It can also process and analyze video in real time. Image recognition with pre-trained models can also be done to classify and identify objects [26–29]. OpenCV is compatible with various programming languages, including Python, C++, and Java, and can run on multiple operating systems such as Windows, macOS, Linux, Android, and iOS. Widely utilized in both academia and industry, OpenCV enables real-time operations through its ability to utilize hardware acceleration, making it suitable for applications in security, medical imaging, and autonomous vehicles. Also, this library is continuously updated by a community of developers, ensuring support for the latest advancements in computer vision technologies.

Although OpenCV is widely used, it can present several issues when used on resource-constrained systems such as the Jetson Nano and Raspberry Pi. These systems have limited computational power, memory, and storage, which makes it challenging to handle the demands of OpenCV, especially for tasks like real-time video processing or deep learning inference. OpenCV often requires custom compilation and careful configuration of dependencies such as CUDA, cuDNN, and TensorRT. Additionally, some OpenCV features may not be fully compatible or optimized for ARM-based processors. Therefore, it is crucial to test OpenCV's functionalities and video processing capabilities before undertaking deep learning operations.

In Figure 5.19, we present a program utilizing OpenCV's video capture. To test this, create a new Python script called "example2.py" within the "chap-jetson" folder, and then run this script from the terminal window as illustrated in Figure 5.24. The program imports the OpenCV library, sets display width and height, and accesses Jetson Nano's USB camera using "VideoCapture(0)" function. It includes a while loop that continuously displays video frames from the camera (Figure 5.18). To run the program, right-click on the VS code editor and select "Run Python File In Terminal"

example2.py

```
# Import library
import cv2

# Set the desired width and height
img_width = 640
img_height = 480

# Initialize video capture using default USB camera (index 0)
captured_video = cv2.VideoCapture(0)

# Start an infinite loop to continuously capture frames
while True:
    # Capture the current frame
    ret, frame = captured_video.read()
    if ret:
        frame = cv2.resize(frame, (img_width, img_height))
    # Display the captured frame in a window named 'my camera'
    cv2.imshow('my camera', frame)
    # Check if the 'q' key has been pressed to break out of the loop
    if cv2.waitKey(1) == ord('q'):
        break
# Release video capture object and close all cv2 windows
captured_video.release()
cv2.destroyAllWindows()
```



Figure 5.24 Running Python program in VS Code to verify video captures using OpenCV.

(Figure 5.13). The program uses the "cv2.imshow()" function to display the current captured frame. If the defined image width and height are different than the default capture, then the frame will be resized accordingly. To exit the video stream or while loop, press the "q" key on the screen, after which OpenCV will close the "my camera" window.

Next, we check OpenCV's video capture frames, where three circular-shaped objects will float around. To make the program more interactive, these objects are made in such a way that they will bounce from the frame boundaries and also from each other. We have created a list for defining circles, and inside that, we have dictionaries to define the center, radius, color, and velocities of the circles. The "update_position" function continuously updates the floating positions of three circles, and the "check_collisions" function verifies the collusion criteria between the shapes and frame boundaries (Figure 5.25).

Figure 5.26 presents the continuous while loop that captures frames and resizes them for display. It uses a for loop to call the update position functions of the circles and then check the collision status. Finally, a circle list is used to define the center positions, radii, and colors.

To run the program, right-click on the VS code editor and select "Run Python File In Terminal." The program uses "cv2.imshow()" function to display the current captured frame. To exit the video stream or while loop, press the "q" key on the screen, after which OpenCV will close the "my camera" window.

example3.py ● example2.py

```
import cv2
import numpy as np

# Set the desired width and height for the video capture
img_width = 720
img_height = 640

# Initialize video capture with the default camera (index 0)
captured_video = cv2.VideoCapture(0)

# Initialize positions, velocities, and radii for the circles
circles = [
    {"center": [100, 100], "radius": 80, "color": (255, 0, 0), "velocity": [4, 3]},
    {"center": [300, 300], "radius": 80, "color": (0, 255, 0), "velocity": [-3, 2]},
    {"center": [500, 500], "radius": 80, "color": (0, 0, 255), "velocity": [2, -4]},
]

# Function to update the position of a circle and check for wall collisions
def update_position(circle):
    for i in range(2):
        circle["center"][i] += circle["velocity"][i]
        if circle["center"][i] - circle["radius"] < 0 or \
            circle["center"][i] + circle["radius"] > (img_width if i == 0 else img_height):
            circle["velocity"][i] = -circle["velocity"][i]

# Function to check for collisions between circles
def check_collisions():
    for i in range(3):
        for j in range(i + 1, 3):
            dist = np.linalg.norm(np.array(circles[i]["center"]) - np.array(circles[j]["center"]))
            if dist < circles[i]["radius"] + circles[j]["radius"]:
                circles[i]["velocity"], circles[j]["velocity"] = \
                    circles[j]["velocity"], circles[i]["velocity"]
```

Figure 5.25 Initialize video by OpenCV, create circle objects and functions for updating circle positions, and verify collisions.

```
# Main loop to capture frames and draw shapes
while True:
    # capture the current frame
    ret, frame = captured_video.read()
    if ret:
        frame = cv2.resize(frame, (img_width, img_height))
    for circle in circles:
        update_position(circle)
    # call the check_collisions function
    check_collisions()
    for circle in circles:
        cv2.circle(frame, tuple(circle["center"]), circle["radius"], circle["color"], -1)
    # Display the captured frame in a window named 'my camera'
    cv2.imshow('my camera', frame)
    if cv2.waitKey(1) == ord('q'):
        break
# Release video capture object and close all cv2 windows
captured_video.release()
cv2.destroyAllWindows()
```



Figure 5.26 Running Python program in VS Code to verify basic image processing and video capture.

5.8 Conclusion

Jetson Nano is a powerful platform for developing high-computational and real-time processing applications in compact and energy-efficient robotic and control projects. Its efficient handling of multidimensional arrays and matrices, along with a wide range of mathematical functions, makes it very useful for low-cost data science, engineering, scientific research, and classroom demonstrations.

In this chapter, we have discussed the key aspects and techniques of configuring Jetson Nano. The initial setup of the Jetson Nano will build a robust groundwork for all subsequent exploration and development efforts, as illustrated in other chapters. We strongly suggest that readers of this book perform real-time package setups and develop programming to validate the OpenCV installation.

5.9 Exercise Problems

(Q1) Prepare a Jetson Nano 4 GB board for deep learning applications by installing the OS image and the necessary libraries, as demonstrated in this chapter.

(Q2) The “example3.py” program in Figures 5.25 and 5.26 demonstrates how to use OpenCV to draw shapes on captured frames. The properties of the circles are defined in a list that is utilized in the “update_position” and “check_collisions” functions. Modify this Python program to create 10 circular shapes, each with a unique color and a radius of 40 pixels. These circles should bounce off the frame boundaries and from one another. The output from your video capture will be similar to Figure 5.27.

(Q3) In this exercise, you’ll alter the code from Q2 to adjust the velocities of the circular shapes. Some will move quickly, while others will move very slowly. The output will resemble Figure 5.21, with the key difference being that some shapes will move and collide more rapidly, and it will affect their velocities.



Figure 5.27 Sample output of Q2. Create 10 circular shapes, each with a distinct color, floating across the video frames.

References

1. Harris, M. (2014). *Jetson TK1: Mobile Embedded Supercomputer Takes CUDA Everywhere*. https://developer.nvidia.com/blog/jetson-tk1-mobile-embedded-supercomputer-cuda-everywhere/.
2. Franklin, D. (2015). *NVIDIA Jetson TX1 Supercomputer-on-Module Drives Next Wave of Autonomous Machines*. https://developer.nvidia.com/blog/nvidia-jetson-tx1-supercomputer-on-module-drives-next-wave-of-autonomous-machines/.
3. NVIDIA (2017). *NVIDIA Jetson TX2 Enables AI at the Edge*. https://nvidianews.nvidia.com/news/nvidia-jetson-tx2-enables-ai-at-the-edge.
4. Gopalakrishna, M. (2018). *Now Available: NVIDIA Jetson AGX Xavier Module for Next-Gen Autonomous Machines*. https://blogs.nvidia.com/blog/nvidia-jetson-agx-xavier-module-now-available/.
5. Franklin, D. (2019). *Jetson Nano Brings AI Computing to Everyone*. https://developer.nvidia.com/blog/jetson-nano-ai-computing/.

6 Villaret, M., Alsinet, T., and Fernández, C. (2021). *Artificial Intelligence Research and Development: Proceedings of the 23rd International Conference of the Catalan Association for Artificial Intelligence*. IOS Press.

7 NVIDIA (2020). *NVIDIA Releases Jetson Xavier NX Developer Kit with Cloud-Native Support*. https://nvidianews.nvidia.com/news/nvidia-releases-jetson-xavier-nx-developer-kit-with-cloud-native-support.

8 Boston Dynamics (2024). *Doing More with Spot*. https://bostondynamics.com/blog/doing-more-with-spot/.

9 ANYbotics (2020). *ANYmal, a Cutting-Edge Legged Robot Platform to Advance Robotics Research*. https://www.anybotics.com/news/advancing-legged-robotics-research/.

10 Alarcon, N. (2019). *Inception Spotlight: New Skydio 2 Drone Powered by NVIDIA Jetson*. https://developer.nvidia.com/blog/skydio-2-jetson-tx2-drone/.

11 Neurala (2016). *Neurala Brings New AI Technology to NVIDIA Jetson Ecosystem*. https://www.neurala.com/press-releases/neurala-brings-new-ai-technology-nvidia-jetson-ecosystem.

12 Aetina (2024). *AN110-NAO Smallest Design with NVIDIA Jetson Nano Module*. https://www.aetina.com/products-detail.php?i=381.

13 Shop, E.R. (2024). *MyCobot 280 Jetson Nano 6 DOF Collaborative Robot (Jetson Nano Version)*. https://shop.elephantrobotics.com/products/mycobot-280-jetson-nano.

14 Nano, J. (2024). *Jetson Nano Technical Specifications*. https://developer.nvidia.com/embedded/jetson-nano.

15 Ssheshadri, P. (2019). *Jetson Nano Developer Kit*. https://developer.download.nvidia.com/embedded/L4T/r32-3-1_Release_v1.0/Jetson_Nano_Developer_Kit_User_Guide.pdf.

16 NVIDIA Developer (2024). *JetPack SDK*. https://developer.nvidia.com/embedded/jetpack.

17 Nvidia (2024). *Introduction to NVIDIA JetPack SDK*. https://docs.nvidia.com/jetson/jetpack/introduction/index.html.

18 NVIDIA Developer (2024). *Get Started With Jetson Nano Developer Kit*. https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#write.

19 NVIDIA Developer (2024). *JetPack Archive*. https://developer.nvidia.com/embedded/jetpack-archive.

20 NVIDIA Developer (2024). *Jetson Download Center*. https://developer.nvidia.com/embedded/downloads.

21 Q-engineering (2021). *Jetson Nano DNN Image*. https://github.com/Qengineering/Jetson-Nano-image.

22 Association, S. (2024). *SD Memory Card Formatter for Windows Download*. https://www.sdcard.org/downloads/formatter/sd-memory-card-formatter-for-windows-download/.

23 EDIMAX (2022). *Support and Services – Driver*. https://www.edimax.com/edimax/download/download/data/edimax/global/download/smb_embedded_wireless_adapter_bluetooth/ew-7611ulb_v2.

24 Franklin, D. (2023). *Jetson-Inference Building the Project from Source*. https://github.com/dusty-nv/jetson-inference/blob/master/docs/building-repo-2.md.

25 Bradski, G. (2000). The OpenCV library. *Dr. Dobb's Journal of Software Tools* 120: 122–125.

26 Sigut, J., Castro, M., Arnay, R. et al. (2020). OpenCV basics: a mobile application to support the teaching of computer vision concepts. *IEEE Transactions on Education* 63 (4): 328–335.

27 Allipilli, H. and Samala, S. (2021). Convolutional neural network and OpenCV based mobile application to detect wear out in car tyres. *Journal of Computing Research and Innovation* 6 (1): 100–113.

28 Dom, C., Heras, J., and Pascual, V. (2017). IJ-OpenCV: combining ImageJ and OpenCV for processing images in biomedicine. *Computers in Biology and Medicine* 84: 189–194.

29 Kadhim, H.J. and Abbas, A.H. (2023). Detect lane line for self-driving car using hue saturation lightness and hue saturation value color transformation. *International Journal of Online and Biomedical Engineering* 19 (16): 4–19.

6

Linux Terminal Overview

6.1 Introduction

The free, open-source operating system kernel Linux was developed by Linus Torvalds in 1991. It was initially developed as a personal project to replace the MINIX OS, and the first version was released under the GNU general public license (GPL), promoting free usage, modification, and distribution. Soon after its development, Linux saw a surge in contributions from a growing community of developers, leading to the emergence of various distributions, or "distros." By the late 1990s, big companies started adopting the Linux kernel with different utilities for their business needs. Major companies such as IBM, Red Hat, and SUSE began adopting Linux for enterprise use. Throughout the 2000s, Linux became the top choice for servers, embedded systems, and supercomputers due to its reliability, security, and affordability. Today, Linux remains a key player in technology, constantly evolving with updates and support from a global community. Its widespread use in cloud computing, data centers, and mobile devices demonstrates its importance and adaptability.

There are several popular Linux-based operating systems (distributions or "distros") that have gained popularity, such as Ubuntu, Fedora, Debian, CentOS, Arch Linux, Mint, Red Hat, etc. Each of these operating systems has its unique features and target audiences. For example, Ubuntu is well-known for its user-friendly interface and regular release cycle. Fedora is often used by developers who prefer testing new features. Debian is known for its stability and serves as the base for many other distributions. CentOS (replaced by Rocky Linux and AlmaLinux) is popular for servers, and Arch Linux is used by advanced users familiar with customization. Other notable distros include openSUSE, Zorin, and Elementary, and each of these has its own strengths.

Linux has some notable advantages over Windows and macOS. As an open-source OS, Linux reduces the financial costs used by a company. Its robust security features and lower malware attacks have made it a safer choice. Also, Linux's flexibility allows users to customize their systems for specific requirements and utilize a wide range of hardware. However, compared to its commercial counterparts, such as Microsoft Windows and MacOS, learning Linux can be challenging. The command-line interface is particularly challenging for beginners and for users familiar with more intuitive interfaces. Sometimes, software compatibility in Linux can be an issue, as some proprietary applications may have limited or no functionality on this platform. Customer support options may also be less comprehensive, as Linux relies primarily on community forums. Despite these challenges, the benefits of Linux continue to make it more attractive to many users and organizations.

This chapter introduces some basic concepts of the Linux terminal and commands, which will be useful for later chapters that have examples of Jetson Nano and Raspberry Pi 5. The Jetson Nano

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

operates on a customized version of Ubuntu Linux known as JetPack, which incorporates support for NVIDIA's compute unified device architecture (CUDA), CUDA deep neural network library (cuDNN), and TensorRT libraries. On the other hand, the Raspberry Pi 5, created by the Raspberry Pi Foundation, runs on Raspberry Pi OS (previously called Raspbian), a Debian-based Linux distribution that is customized for Raspberry Pi's lightweight hardware.

6.2 Basic Terminal Commands and Syntax

The default command-line shell and programming language for most of the Linux distributions is called Bourne Again SHell (Bash). It is an updated version of the original Unix Bourne shell (sh) that incorporates features from other popular shells and is used through a Linux terminal application. The terminal is a powerful command-line interface software that allows users to interact directly with the operating system and execute various commands. Commands in the terminal have some specific syntax, and they can read text inputs and output text to the terminal. For example, they begin with the command name itself, followed by an option (also known as flags) and arguments. Users can have just a command or a command with one or more options or a command with one or more arguments.

Figure 6.1 shows how Linux commands, options (or flags), and arguments can be used to modify the behavior of commands. For example, the "cd" command stands for change directory, and it can be used with an argument specifying the folder name to move into (e.g. /home/user/Desktop). The "cp" command with the option "r" can copy a directory to another directory (e.g. /home/user/Documents to /home/user/Desktop) with all the contents. Inside a folder the "ls" command can be used to list all the files that the folder contains. The "-l" option can be used to list files in long formats that provide detailed information about each file. The "sudo" (superuser do) command enables authorized users to run commands with elevated privileges as specified by the security policy. It is commonly employed for administrative tasks that demand higher access levels, such as software installation, system configuration changes, and account management. In Figure 6.1, "sudo" uses "-u" option for the user "user" to list the files in the "bin" folder. The "touch" command is used to create empty files, and if used with "-a" option, the terminal will update the access time of a file (e.g. data.txt) without modifying its content. This can be checked in the terminal by executing the "stat" command (e.g. stat data.txt). "rm" command can be used to remove (delete) files and directories, and if used with "-f" option, it will forcefully remove the specified file (e.g. test.txt). In Figure 6.1,

Command	Options	Arguments
cd		/home/user/Desktop
cp	-r	/home/user/Documents /home/user/Desktop
Is	-l	
sudo	-u	user Is /bin
touch	-a	data.txt
rm	-f	/home/user/Desktop/test.txt

Figure 6.1 Arrangement of Linux commands, options (or flags), and arguments in the terminal.

replace the "user" with your user name. For example, in the Jetson Nano setup (Chapter 5), our user name was "jetson," therefore, the second row of Figure 6.1 will be "cp -r /home/jetson/Documents /home/jetson/Desktop."

There are many other Linux commands, options, and arguments available. In this chapter, we will demonstrate some basic commands that will be utilized for software installation or other common tasks in Jetson Nano or Raspberry Pi. For a comprehensive list of Linux commands, it is advisable to consult the Linux Commands Handbook [1–3]. Learning Linux commands offers numerous benefits that can significantly enhance the efficiency and effectiveness of an embedded system engineer.

6.3 Overview of File System

The Linux and Windows file systems have different structures. Linux uses a single root directory from which all other directories and files originate. On the other hand, Windows starts with drive letters (e.g. C:\, D:\), each representing a separate file storage system. Linux's permissions model is also different. It is based on user, group, and others, with different levels of read, write, and execute permissions. File path separators also differ, with Linux using forward slashes (/) and Windows using backslashes (\). In Linux, hidden files are simply prefixed with a dot, while Windows uses a hidden attribute. Software installation process in Linux relies on package managers and centralized repositories, whereas Windows uses executable installers or its Microsoft Store. The structure of system directories also varies, with Linux using paths such as "/bin" and "/home" and Windows using "C:\Windows" and "C:\Program Files". These differences demonstrate the unique design approaches of the two operating systems. Linux typically aims for a unified, flexible, and highly configurable structure, and Windows prioritizes interactive and straightforward drive-based organization. Each approach has its merits, focusing on different user needs.

6.4 Navigating Files and Directories

In the command-line environment, understanding the file system and the hierarchy standard is essential before navigation. This can be helpful for referring files or directories within inputs and outputs in the commands. Exploring the file system in Linux starts with the root directory, represented by a slash (/), which is the highest level in the hierarchy. Everything else, such as directories, external drives, and network shares, are represented within this single file system. The home directory is where user-specific files are stored, with each user having a personal directory named after their username. Also, the home directory in Linux is accessible with the tilde (~) character. Ubuntu Linux distributions typically include directories such as Desktop, Documents, Downloads, Music, Public, Videos, and Pictures within each user's home directory. The root directory contains the root user's home folder, distinct from the file system root. Other important directories include "etc" (configuration files), "bin" (binary executable), "sbin" (administration binaries or programs), "lib" (shared libraries), "mnt" (mounted file systems), "media" (removable file systems), etc. Additionally, "dev" holds references to hardware, "proc" contains process information, and "sys" holds kernel parameters. Normal users generally lack permission to alter system folders, making it safe to look around. To check all available directories in the root folder, we can use the command "ls /" in the terminal. Figure 6.2 shows the output of "ls /" in the Jetson terminal.

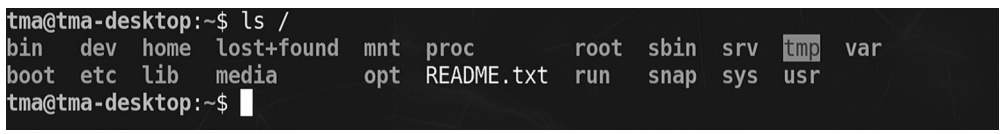


Figure 6.2 Using "ls /" to check all available directories and files in the root from the terminal.

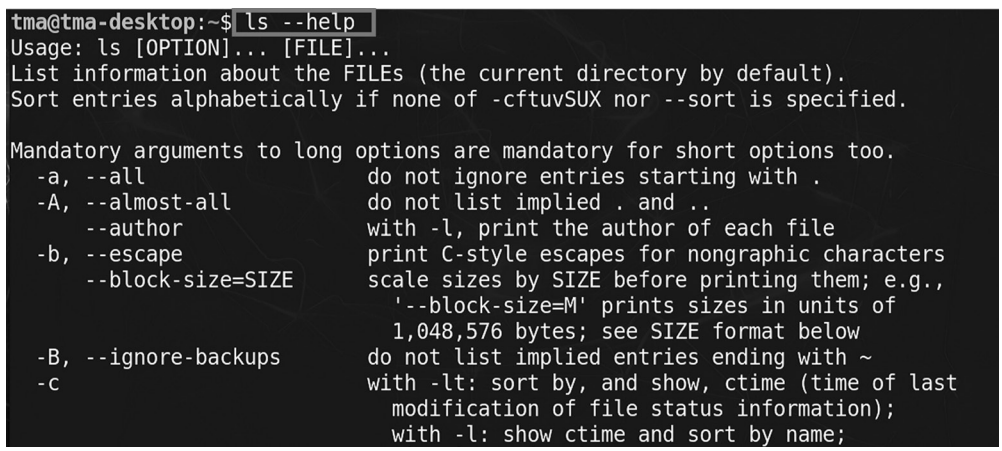


Figure 6.3 Using "ls --help" to check the summarize function of the "ls" command.

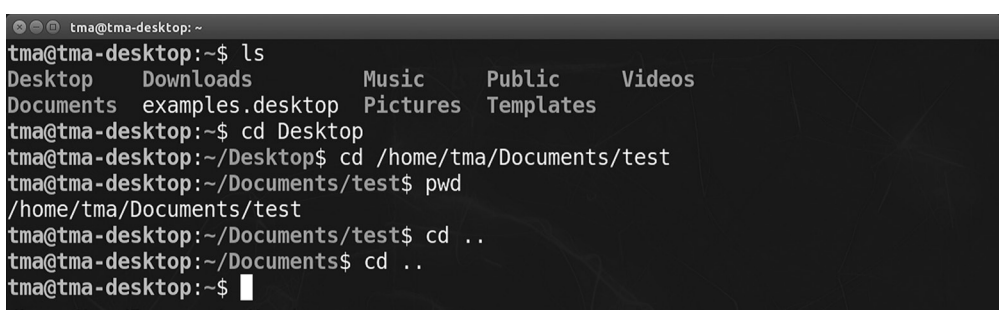


Figure 6.4 Listing available directories, accessing directories using location, checking the path of current directory, and navigating backward to a parent directory.

Note that the username (tma) in this chapter's figures differs from the one used in Chapter 5. This is because we used a different Jetson Nano 4 GB device to generate the images for this chapter. However, this does not affect the Linux terminal commands or functionality, which would remain the same had we used the Jetson Nano board set up in Chapter 5.

To check the functionality of a command and a concise summary, the "--help" command can be used. For example, typing "ls --help" in the terminal will show a description of the "ls" command without accessing the full manual page (Figure 6.3). It should be noted that the "man" (sort for manual page) provides a more detailed description of a command, but it is not included with the default JetPack installation to save memory.

If we want to access certain folders from the terminal, we can use the command "cd" (stands for change directory). Figure 6.4 shows that if "cd Desktop" is used, the directory will change to the

user's "Desktop" directory. To access a different directory, the location can be used after the "cd" command (e.g. cd /home/tma/Documents/test), and the "pwd" (print working directory) command can be used to display the current directory path in which the user is currently working. This command is useful for verifying the location within the file system hierarchy. The "cd .." command can be used to navigate to the parent directory (go backward in the hierarchy) of the current working directory.

We might have a situation when the folder name has a space. In that case, Linux will return an error and will be unable to access the folder if the folder name is entered directly. To address this, a backward slash (\) can be used before the space, indicating to Linux that there is a space in the name. For example, if there is a folder named "car data" inside "Documents," the address should be written as "car\ data" to access that folder. This example is shown in Figure 6.5.

In Linux, the root directory is represented by a single forward slash (/) and serves as the top-level directory in the file system hierarchy, with all other directories and files branching from it. Accessing this directory is done using the "cd /" command, as illustrated in Figure 6.6. When navigating to the root directory, the tilde symbol (~) will change to a forward slash (/). If we check the root directory using "ls," it will display various system configuration and administration files. We can go back to the user's home directory from the root by using the "cd" command and location.

If we would like to move to the user's home directory quickly, the "cd ~" command is utilized, as shown in Figure 6.7. Here, the tilde symbol (~) serves as a convenient shortcut, representing the path to the current user's home folder.

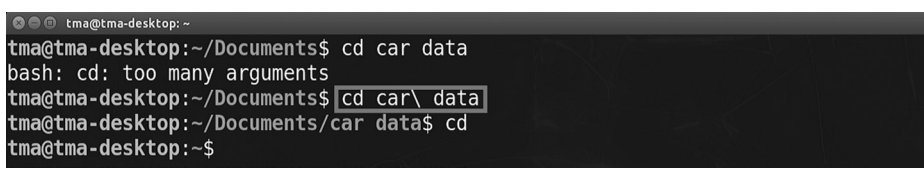


Figure 6.5 Using backslash (\) to access a directory that has space.

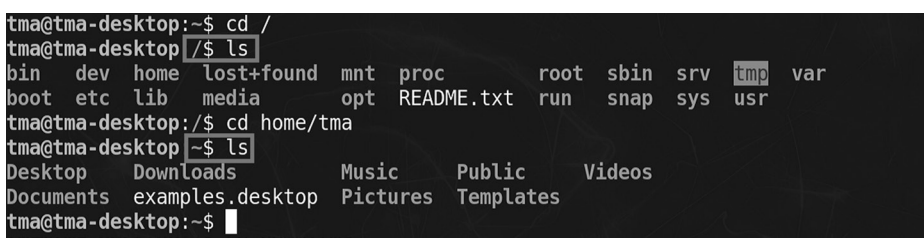


Figure 6.6 Navigate to the root directory and its folders, then return to the home directory.

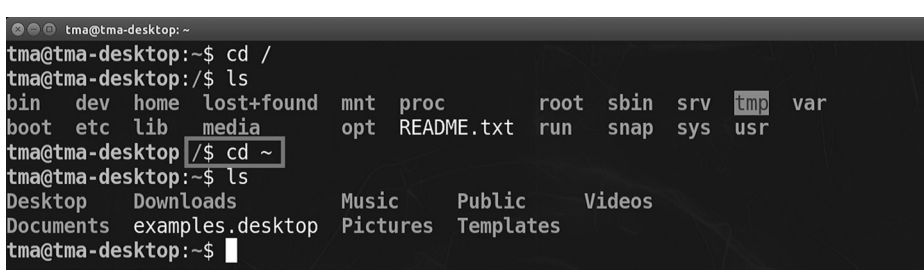


Figure 6.7 Navigate to the root directory using "cd /" and return to the home directory using "cd ~."

6.5 Create, Edit, and Delete

To create a new folder from the terminal, the "mkdir" (make directory) command is used. After opening a terminal, we can use the "mkdir" command followed by the desired folder name. For example, "mkdir test2" will create a directory named "test2" in the current working directory. Figure 6.8 shows if we execute the command from the "Documents" folder, it will create a new folder ("test2") inside "Documents."

The "mkdir" command can also be utilized to create multiple directories simultaneously by separating their names with spaces. To create multiple directories within a specific directory, we need to include the parent directory name in the path. For example, "mkdir test2/data test2/data1 test2/data2" creates three folders (data, data1, and data2) inside the "test2" directory. This process is illustrated in Figure 6.9.

Let's say we would like to create a directory "data1" inside "test3." However, the "test3" is not created in the current "Document" directory. In this scenario, the "mkdir" command will return an error. In this case, to create multiple nested directories in one command, the "-p" option can be used as "mkdir -p parentfolder/childfolder." This will create both parent and child folders within it if they do not already exist. The process is illustrated in Figure 6.10. The "mkdir" command is a

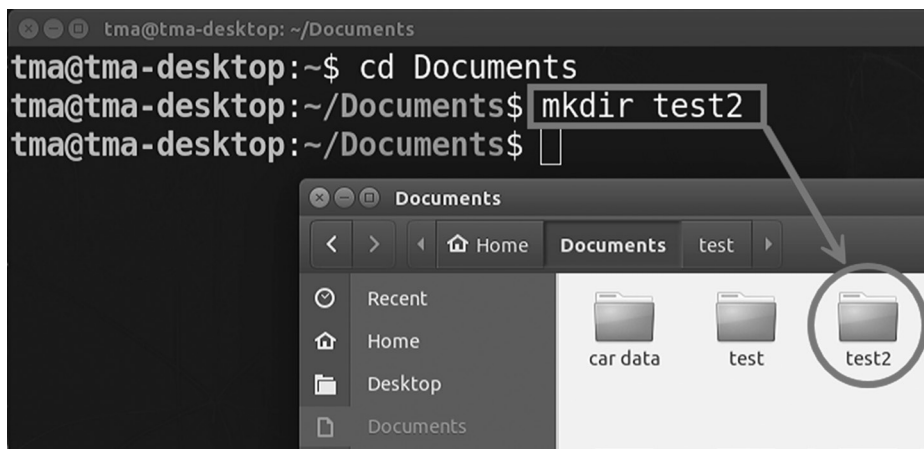


Figure 6.8 Use "mkdir" command and directory name to create a new directory in the current folder.

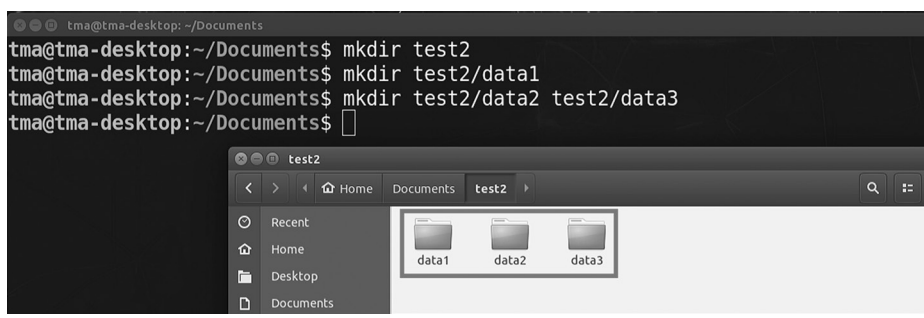


Figure 6.9 Use "mkdir" to create multiple directories (data, data1, and data2) simultaneously inside a directory (test2).

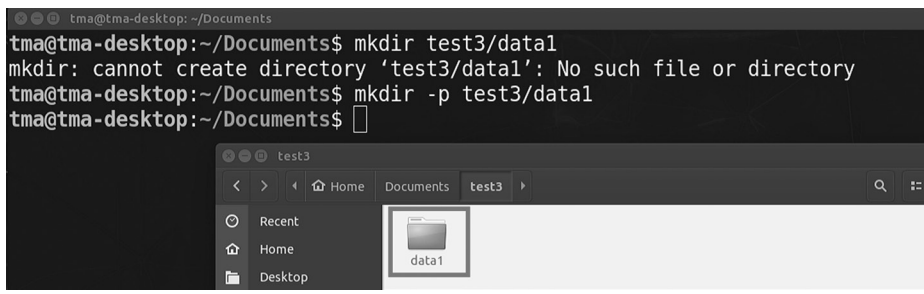


Figure 6.10 “mkdir” command with “-p” option to create multiple nested directories in the current directory.

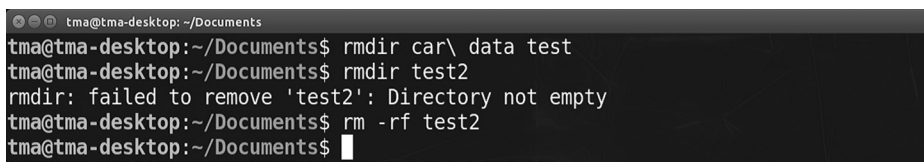


Figure 6.11 Using “rmdir” command to delete an empty directory and “rm” command to delete a directory that has files or subdirectories.

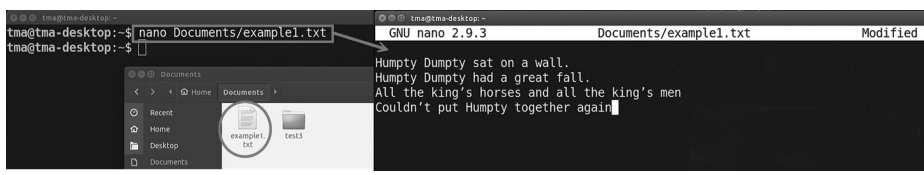


Figure 6.12 Create a text file and write on it using the GNU Nano editor.

straightforward and effective tool for directory management in Linux. It works seamlessly with scripts and facilitates the automation of complex directory structures.

The “rmdir” command in Linux is used to remove empty directories from the file system. It acts as a safeguard against unintended data loss by ensuring that only directories without any files or subdirectories can be deleted. To use this, the “rmdir” command is followed by the name of the directory to be removed. If the directory is not empty, the command will not execute and will return an error. For example, Figure 6.11 shows that “rmdir” can remove the “car data” directory. However, when it tries to delete the “test2” directory, it fails as it contains other directories inside. This command is particularly useful for cleaning up empty directories and maintaining an organized file system. If we need to delete directories and all of their subdirectories or files, we may use “-rf” option with the “rm” command. The “-r” (or recursive) option allows the command to delete the directory that has files or subdirectories, and the “-f” (or force) option bypasses any warnings. We may use these two options together (“-rf”) with the “rm” command. Since “-rf” permanently deletes the directory and its contents without recovery options, we need to be careful while using this option. Figure 6.11 shows how we can remove an empty directory and a directory with subdirectories and files.

Figure 6.12 demonstrates the process of creating a text file using the Nano text editor. To install Nano on a Linux system, execute the command “sudo apt-get install nano” in the terminal. Make

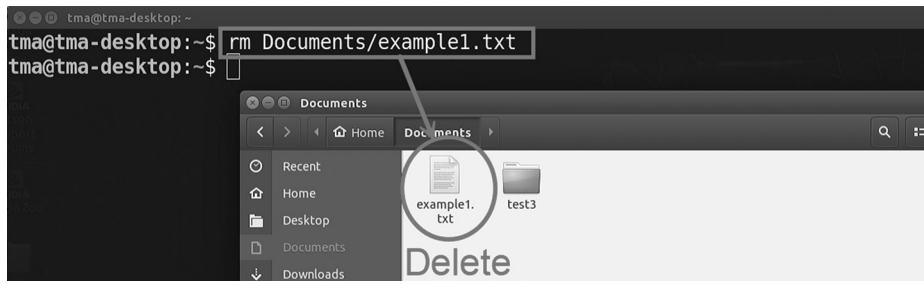


Figure 6.13 Using "rm" command to delete "example1.txt" file from the "Documents" folder.

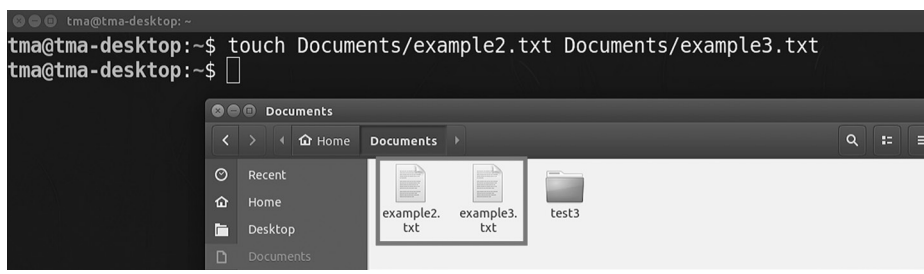


Figure 6.14 Using "touch" command to create empty text files.

sure that Jetson is connected to the internet before proceeding with the installation. Nano editor in Linux is simple and user-friendly, and it is designed for command-line interface environments. It allows users to create, edit, and manage text files directly from the terminal. To create a file with Nano, type "nano filename" in the terminal. Figure 6.12 shows that to create a text file named "example1.txt," the command "nano Documents/example1.txt" is entered. This command is executed from a directory other than "Documents," so specifying the location ("Documents") before the text file name is necessary. Nano will then create the new text file "example1.txt" and open it in the GNU Nano text editor. Here, we can write the text file (e.g. Humpty Dumpty rhyme) using the editor. To save and close the file, press "Ctrl+x," then "y," and finally, "Enter."

To delete the file, we can simply use the "rm" command and file name as shown in Figure 6.13.

The "touch" command can be used to create new or empty files. While "nano" launches an interface for text input and editing, the "touch" command simply creates the file without opening it for editing. Figure 6.14 shows how the "touch" command can be used to create two text files (example2.txt and example3.txt) inside "Documents."

6.5.1 Copy and Move Operations

To copy files and directories from one location to another, "cp" command can be used. Figure 6.15 shows an example of using "cp," where an existing "example1.txt" source file is copied to "example2.txt" inside the "Document" directory.

Let's say there are subdirectories "test3/data/newfile" within "Document," and we need to copy "example1.txt." If no name is specified for the copied file, it will retain the same name as the source file. However, if a new name (e.g. "new.txt") is assigned, the source file will be copied with the new name. This example is shown in Figure 6.16.

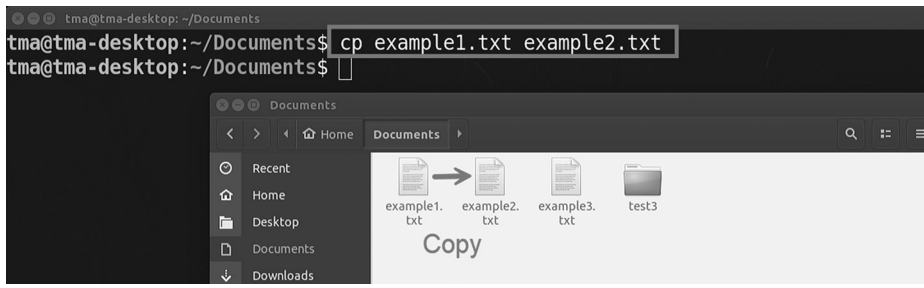


Figure 6.15 Using "cp" command to copy a file "example1.txt" into the current "Document" directory.

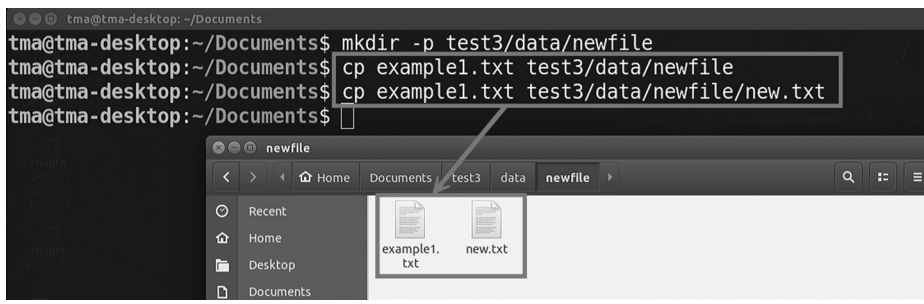


Figure 6.16 Using "cp" command to copy files while retaining the original name (example1.txt) or assigning a new name (new.txt).

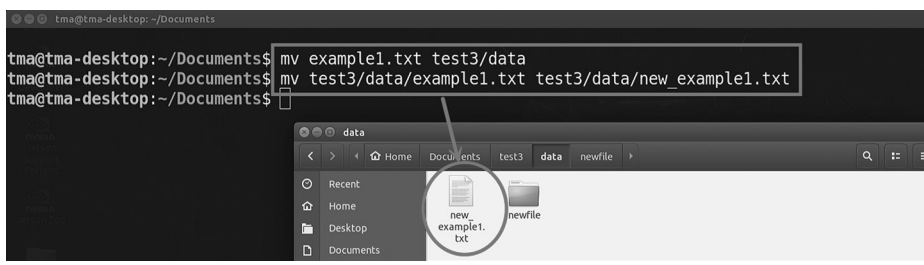


Figure 6.17 Using "mv" command to move and rename a text file.

The "mv" command is used to move or rename files and directories. For example, "mv example1.txt test3/data" moves the source file "example1.txt" to the specified "test3/data" directory. Unlike Nano (text editor for creating and modifying file contents), "mv" doesn't alter the file's content. Instead, it changes the file's location or name and assists in organizing the file system. Figure 6.17 shows these examples. In the first row, the "mv" command moves the "example1.txt" file inside the "data" folder, and in the second row, the "mv" command not only moves the "example1.txt" but also renames it (new_example1.txt).

6.6 Create and Execute Python Code from Terminal

In the Linux terminal, Python programming files can be created, edited, and executed. To create a Python file, the ".py" extension is used. Figure 6.18 demonstrates the creation of a Python file named "python_example1.py" using the "touch" command and editing it using the Nano editor.

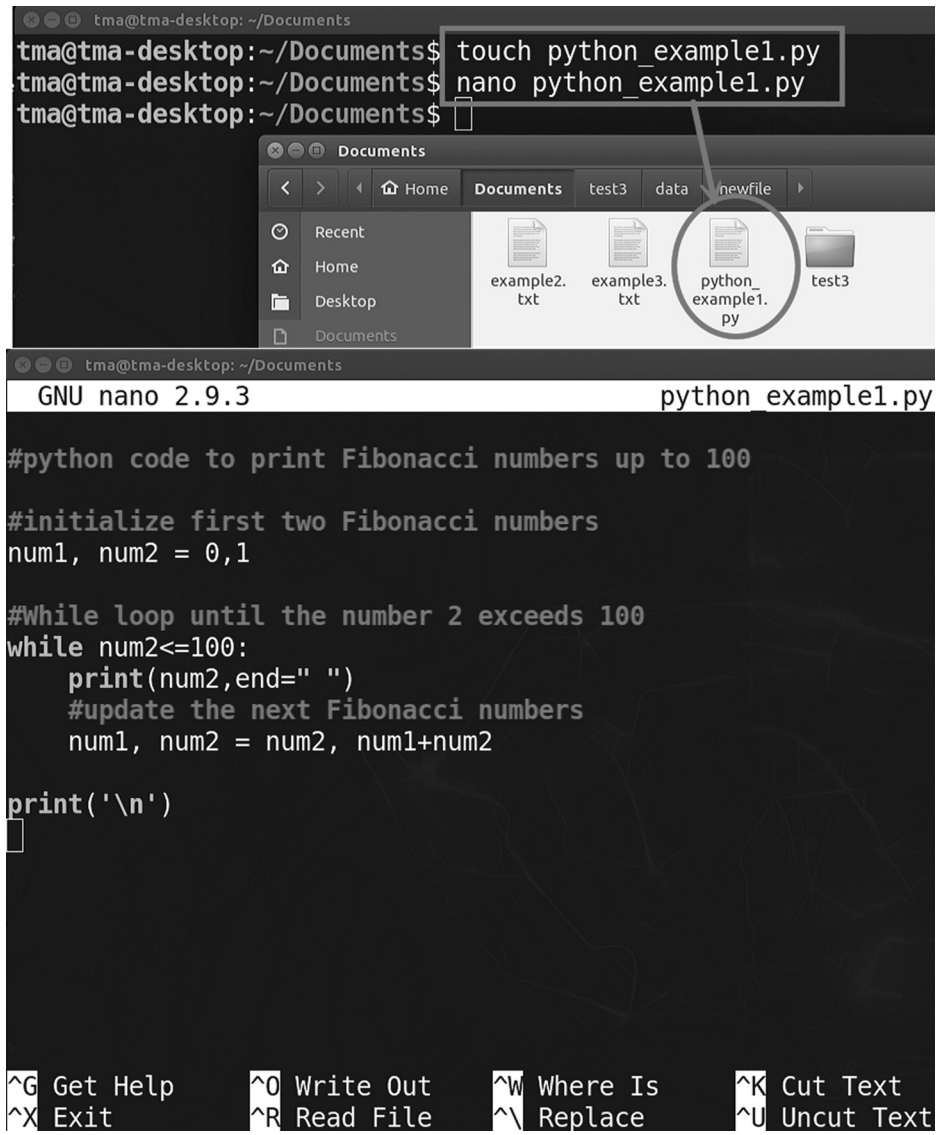


Figure 6.18 Create a Python file with the "touch" command and edit it using the Nano editor.

Let's say we would like to create a Python program that prints all Fibonacci numbers up to 100. In that case, we can write, modify, or copy-paste the program in the Nano editor. Once finished, save and exit Nano by pressing "ctrl+x," "y," and "Enter."

Now, to run the Python code from the terminal, we can use the command "python3" command followed by the Python file name. This command will run the code using the Python3 interpreter, and the program output will be displayed in the terminal, as shown in Figure 6.19. In the Jetson Nano, if we use the "python" command, the code will run through Python 2 as it is the default version of Python for Jetpack 4.6.

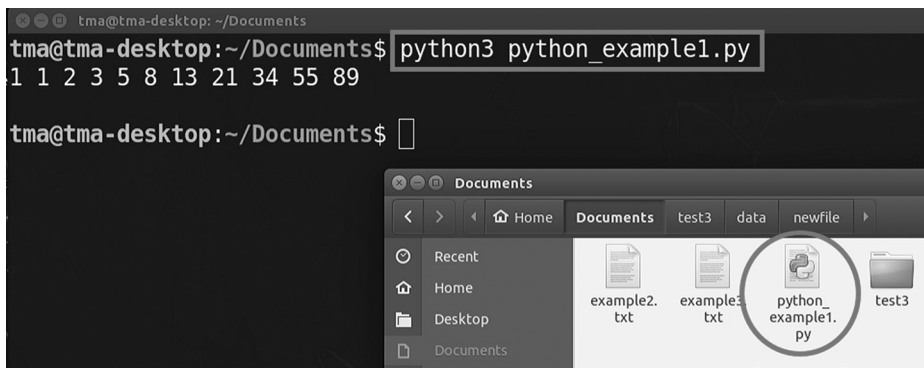


Figure 6.19 Run Python code from the terminal using "python3" command.

6.7 Common Wildcard Characters

Wildcard characters are used in shell commands to match filenames and directory patterns. The most commonly used wildcards are the asterisk (*) (represents zero or more characters), the question mark (?) (represents exactly one character for one "?"), square brackets ([ ]) (match any single character within), hyphen (-) (define character range, such as [a-c].txt), the caret (^) or exclamation mark (!) within square brackets (when used at the beginning, matches any character not enclosed), and braces ({ }) are used to specify a comma-separated list of alternatives (file{1,2,3}.txt matching file1.txt, file2.txt, and file3.txt). These wildcards provide flexible options for file matching and manipulation in the terminal. In this section, we will discuss the "*" and "?" wildcard characters.

The asterisk (*) acts as a wildcard for pattern matching, representing any number of characters in file or directory names. For example, "ls *.py" will list all files with a ".py" extension, and "cp example* /data" will copy all files starting with "example" to the "/data" directory. Another character is the dot (.), which can indicate the current directory and mark hidden files or extensions. Dot "." can be used in commands such as "./script.sh" to run scripts in the current directory. Figure 6.20 shows how we can move all the files and folders from the "test3/data/" to the current "Document" folder.

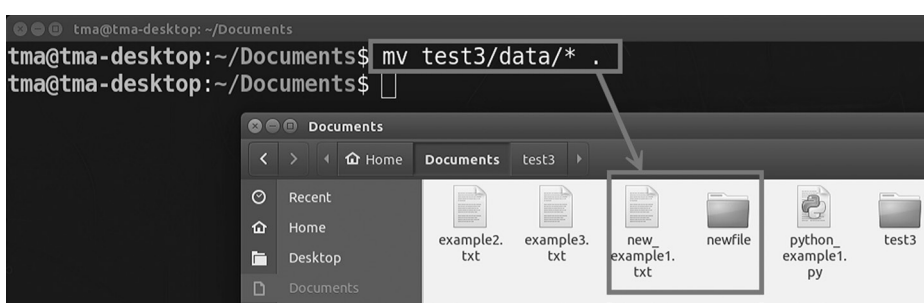


Figure 6.20 Moving "newfile" and "new_example1.txt" from "test3/data" directory to the current (Documents) directory using ".".

```
tma@tma-desktop: ~/Documents
tma@tma-desktop:~/Documents$ rmdir test3
rmdir: failed to remove 'test3': Directory not empty
tma@tma-desktop:~/Documents$ rm -rf test3
tma@tma-desktop:~/Documents$ rm example?.txt
tma@tma-desktop:~/Documents$
```

Figure 6.21 Using "rm" command to delete all files and folders in "test3," and "rm" with "?" to delete all files that start with "example," have a ".txt" extension, and in between have only one character.

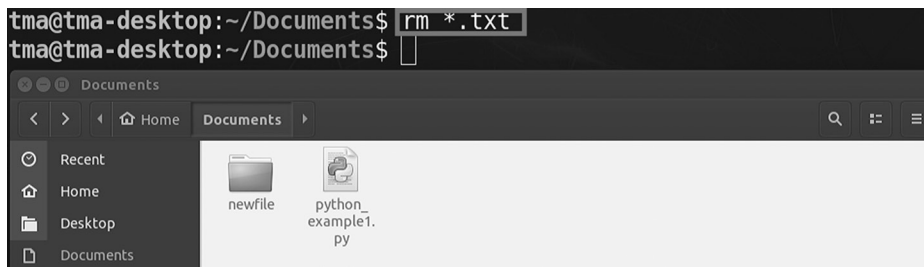


Figure 6.22 Delete all files that have ".txt" extension from the "Document" folder.

We have seen in Sections 6.5 and 6.6 that "rmdir" and "rm" commands can be used to delete directories and files. To remove files that share common characters, such as "exampleA.txt," "example1.txt," "example2.txt," and "example3.txt," the wildcard character question mark (?) can be used. Figure 6.21 demonstrates that to delete the "test3" directory along with all its subdirectories and files, the "rm" command with the "-rf" option is used. To delete all files that start with "example," have a ".txt" extension and in between have exactly one character, the "?" symbol can be used for pattern matching. To get files with multiple characters in between, we can use "?" multiple times. For example, "example??.txt" can be used for "example12.txt," "example45.txt," and "exampleAB.txt" files.

To delete all files with a ".txt" extension, the "*.txt" pattern can be used. Figure 6.22 shows how all text files in the "Document" folder can be deleted using the "rm" command.

6.8 Find, View, and Get Information

The "find" is a powerful tool for locating files and directories within a specified directory. It works by recursively exploring directories from a given starting path and identifying files that match specific criteria such as name, size, type, modification date, or permissions. In Figure 6.23, "find . -name "*.txt"" searches for all ".txt" files in the directory and its subdirectories. The "find" command can also execute other actions on the matched files, such as deleting them with "-delete" or executing other commands using "-exec." For example, in Figure 6.23, to delete all ".txt" files from the current and subdirectories, "find . -name ".txt" -delete" can be used.

Figure 6.24 illustrates the use of the "find" command to search for all files starting with "example" in the current directory and its subdirectories. It also shows that the command can be used to search for files and directories containing the letter "w."

Figure 6.25 shows how "find" can be used to search files or folders starting with "b" in the "Desktop." Here, from the "Document" folder, "~/" can be used to access the current user's home directory. "find" can also search for files based on their memory size. For example, as shown in Figure 6.26,

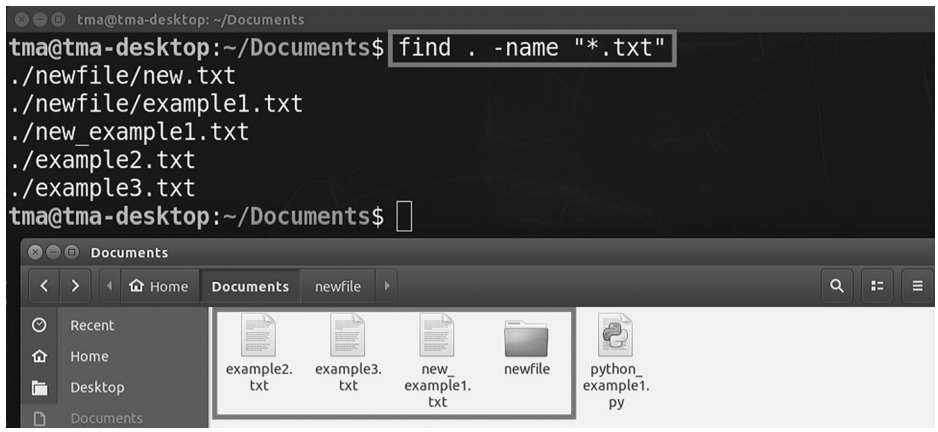


Figure 6.23 Using "find" command to search for all ".txt" files in the current directory and its subdirectories.

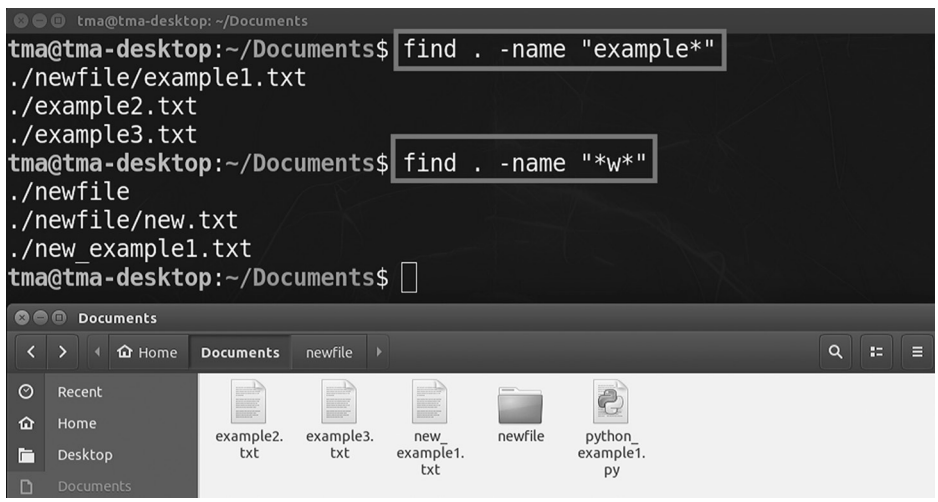


Figure 6.24 Using "find" command to search for all files and folders starting with "example" and having the letter "w" in their name.

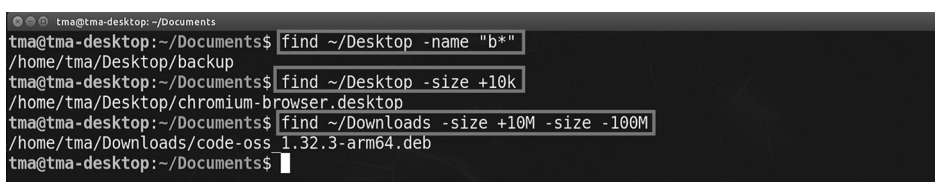


Figure 6.25 Using "find" to search for names starting with "b" on the "Desktop" and search for files within a specified memory size range.

the command searches for files on the "Desktop" that are larger than +10 kB (10 kilobytes). It is also possible to define a size range, for example, finding files greater than +10 MB (10 Megabytes) and less than -100 MB (100 Megabytes) in the "Download" directory.

Now, suppose we want to save all the file names of the "Document" directory into a text file. We can achieve this by using the redirection operator (>) with the "ls" command. This operator will redirect

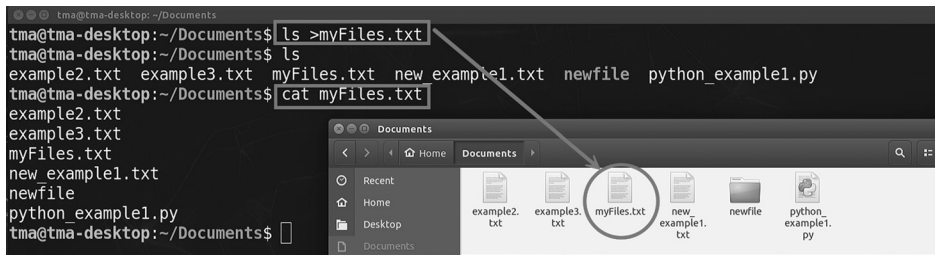


Figure 6.26 Write outputs into a text file using the redirection operator (>) and display the contents of the text file using "cat" command.

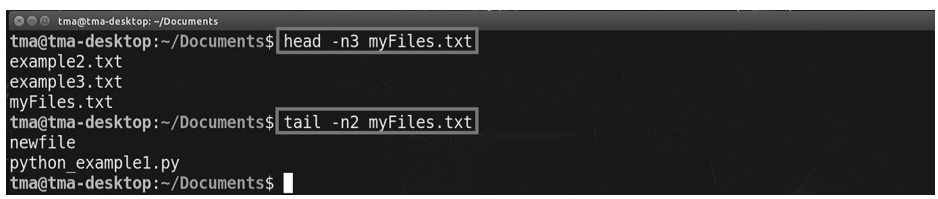


Figure 6.27 View the first three lines and last two lines of the "myFiles.txt" using the "head" and "tail" commands, respectively.

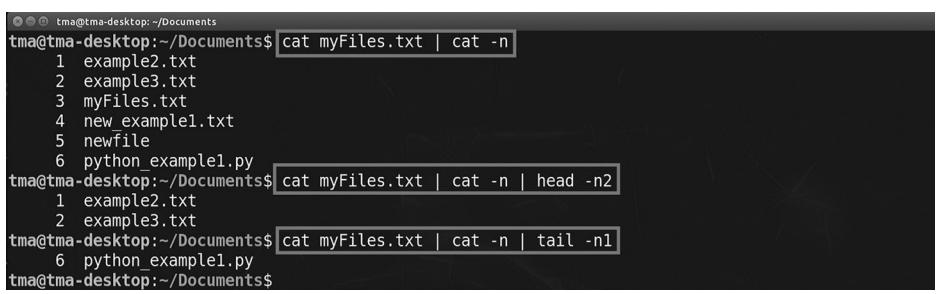


Figure 6.28 View text file contents with line numbers and view the first two or last one lines using "head" and "tail" commands.

the output of "ls" to the specified text file (e.g. myFiles.txt). To view the contents of the text file from the terminal, we can use the "cat" (stands for concatenate) command, as shown in Figure 6.26.

In the example text file "myFiles.txt," there are only six lines. However, if there are numerous lines, the "head" command can be used to view only the first few lines (ten lines by default). To specify a different number of lines, the "-n" option can be used. For example, in Figure 6.27, "head -n3" displays the first three lines. Similarly, to view the last few lines of a file, the "tail" command can be used with the "-n" option. For example, "tail -n2" will show the last two lines.

The pipe (|) command in the terminal allows the output of one command to be used as input for another command. This enables users to create complex and efficient workflows by linking simple commands together. For example, as shown in Figure 6.28, the command "cat myFiles.txt | cat -n" lists file names from "myFiles.txt" and numbers them using "cat -n." These numbered lines can also be passed to the "head" or "tail" commands with "-n" option to display the first or last few lines, respectively.

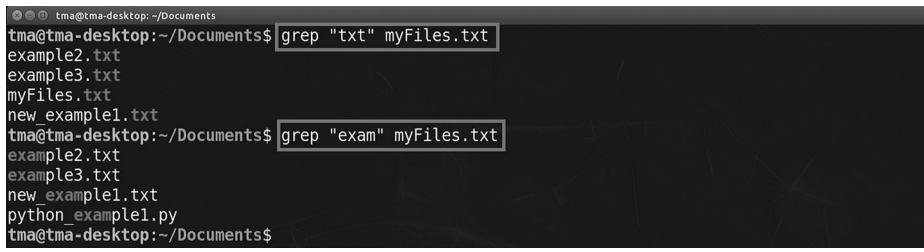


Figure 6.29 Search the terms "txt" and "exam" in "myFiles.txt" using "grep."

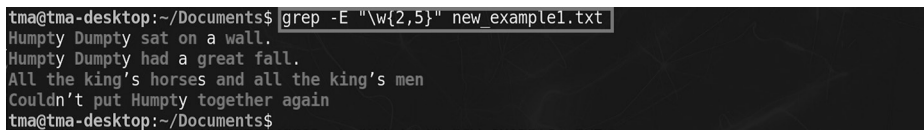


Figure 6.30 Find words with 2–5 characters in "new_example1.txt" file using the "\w," curly bracket ({}), and extended "grep" command.

Another powerful text-search utility in Linux is "grep," which is used to find patterns within files or output from other commands. The name stands for "Global Regular Expression Print" and describes its function of searching global text patterns using regular expressions. "grep" scans files or input line by line and displays lines that match the specified pattern. For example, "grep "txt" myFiles.txt" in Figure 6.29 searches for the term "txt" within the file named "myFiles.txt" and shows all matching lines with the term highlighted. It can also be used with various options to refine searches, such as "-i" for case-insensitive searches, "-r" for recursive directory searches, and "-v" to show non-matching lines. The filtering and extracting capabilities of specific data from large text volumes make "grep" an important tool for system administrators and developers working in Linux environments.

The "grep" command has versatile applications for advanced text processing. When used with the "-E" (extended) option, it allows for more advanced and flexible pattern-matching operations. To test this, create a text file named "new_example1.txt" in the "Documents" directory, write any content (e.g. nursery rhymes in Figure 6.30), and save the file. To find words with 2–5 characters using the "\w" and curly bracket ({}) metacharacter, the input in the terminal would be "grep -E "\w{2,5}" new_example1.txt" (Figure 6.30). "grep" can also be used with "+," "?," "|," and "()" metacharacters to perform various complex search operations.

6.9 Permission and Ownership

File permission and ownership in Linux are critical for managing security and access control. Each file and directory has an owner (usually its creator) and is associated with a specific group. Permissions dictate who can read, write, or execute a file and are categorized into owner (user), group, and others. These permissions are represented by characters for read (r), write (w), and execute (x). For example, "rwxr-xr--" means the owner (first three characters) has full permissions, the group (characters four to six) can read and execute, and others (last three characters) can only read.

Ownership and permissions can be updated using "chown" (change ownership) and "chmod" (change mode). When a user tries to access a file, the system checks its permission mode to determine if the user has the necessary rights.

6.9.1 Permissions Using Octal and Symbolic Notations

User permissions can be modified using the "chmod" command, which can use either octal notation (e.g. chmod 755) or symbolic notation (e.g. chmod u+rwx, g+rx, o+rx). File ownership changes can be made with "chown" and "chgrp" commands. In octal notation, values of read (4), write (2), and execute (1) are combined. For example, "chmod 644" is equivalent to "rw-r--r--." On the other hand, symbolic notation allows permissions to be added (+), removed (-), or set (=) specifically for the user (u), group (g), others (o), or all (a). In Linux operating systems, proper use of these commands is critical so that file access and modifications are limited to authorized users, thereby maintaining system security. Figure 6.31 shows the symbolic

Octal permissions: read (r) = 4, write (w) = 2, execute (x) = 1

user	group	others	chmod	description
- - - (0+0+0 = 0)	- - - (0+0+0 = 0)	- - - (0+0+0 = 0)	000	No permissions for anyone.
r w – (4+2+0 = 6)	- - - (0+0+0 = 0)	- - - (0+0+0 = 0)	600	Read and write permissions for user only.
r w – (4+2+0 = 6)	r - - (4+0+0 = 4)	r - - (r+0+0 = 4)	644	Read and write permissions for user. But only read for group and others.
r w – (4+2+0 = 6)	r w – (4+2+0 = 6)	r w – (4+2+0 = 6)	666	Read and write permission for all.
r w x (4+2+1 = 7)	- - - (0+0+0 = 0)	- - - (0+0+0 = 0)	700	Read, write, and execute permissions for user only.
r w x (4+2+1 = 7)	- - x (0+0+1 = 1)	- - x (0+0+1 = 1)	711	Read, write, and execute permissions for user. But only execute for the group and others.
r w x (4+2+1 = 7)	r – x (4+0+1 = 5)	r - - (4+0+0 = 4)	754	Read, write, and execute permissions for users. But read and execute for the group, and only read for others.
r w x (4+2+1 = 7)	r – x (4+0+1 = 5)	r – x (4+0+1 = 5)	755	Read, write, and execute permissions for users. But read and execute for the group and others.
r w x (4+2+1 = 7)	r w x (4+2+1 = 7)	r w x (4+2+1 = 7)	777	Read, write, and execute permissions for everyone.

Figure 6.31 Symbolic notations for user, group, and others, and corresponding "chmod" octal notations with description.

notations for the user, group, and others, along with their corresponding octal notations, which can be used with the "chmod" command.

6.9.2 Update Permission Using Operators

Symbolic permission can use "+," "-," "=" operators to provide a straightforward and intuitive way to control access. The "+" operator adds the specified permissions to the existing ones. For example, "chmod g+x myFiles.sh," adds execute permission to the group without altering other permissions. The "-" operator removes the specified permissions. "chmod g-r myFiles.sh" removes read permission from groups while leaving other permissions unchanged. The "=" operator sets the permissions explicitly by updating existing permissions. For example, "chmod g=rx myFiles.sh" sets the group permissions to read and execute. These operators provide precise and flexible control over file and directory access in Linux. Figure 6.32 shows how the original permissions can be modified using the symbolic permission operation.

Now, let's look at an example of an executable Python file and how its permissions can be updated for different groups. First, create a Python file in the terminal using the "touch" command, and then use the Nano editor to write a simple code that prints a statement. To make the Python file executable, add "#!/usr/bin/env python3" at the beginning of the script (Figure 6.33). This line instructs the system to use the Python interpreter from "bin" to run the script. After writing the Python code, save and close the Nano editor by pressing "Ctrl+x," "y," and "Enter."

To obtain detailed information on test.py and its current permission level, we can use the "stat" command. This tool provides comprehensive information about files and file systems

Update permission groups: u (user), g (group), o (others), and a (all)
"+" add permission, "-" remove permission, "=" reset to new mode

Octal	Symbolic	Update using "+", "-", and "=" operators	Resultant permission
644	rw-r--r--	o-r	rw-r----- (640)
644	rw-r--r--	u+x	rwxr--r-- (744)
644	rw-r--r--	g+x	rw-r-xr-- (654)
664	rw-rw-r--	a=r	r--r--r-- (444)
664	rw-rw-r--	a+rwx	rwxrwxrwx (777)
664	rw-rw-r--	u=r, g=r, o=	r--r----- (440)
755	rwxr-xr-x	u-x	rw-r-xr-x (655)
755	rwxr-xr-x	u-w	r-xr-xr-x (555)
755	rwxr-xr-x	o-r	rwxr-x--x (751)
777	rwxrwxrwx	o=r	rwxrwxr-- (774)
777	rwxrwxrwx	o=	rwxrwx--- (770)
777	rwxrwxrwx	g-rx	rwx-w-rwx (727)

Figure 6.32 Update file permission for "user," "group," "others," and "all" using "+" operator to add permission, "-" operator to remove permission, and "=" operator to reset permission to the new mode.

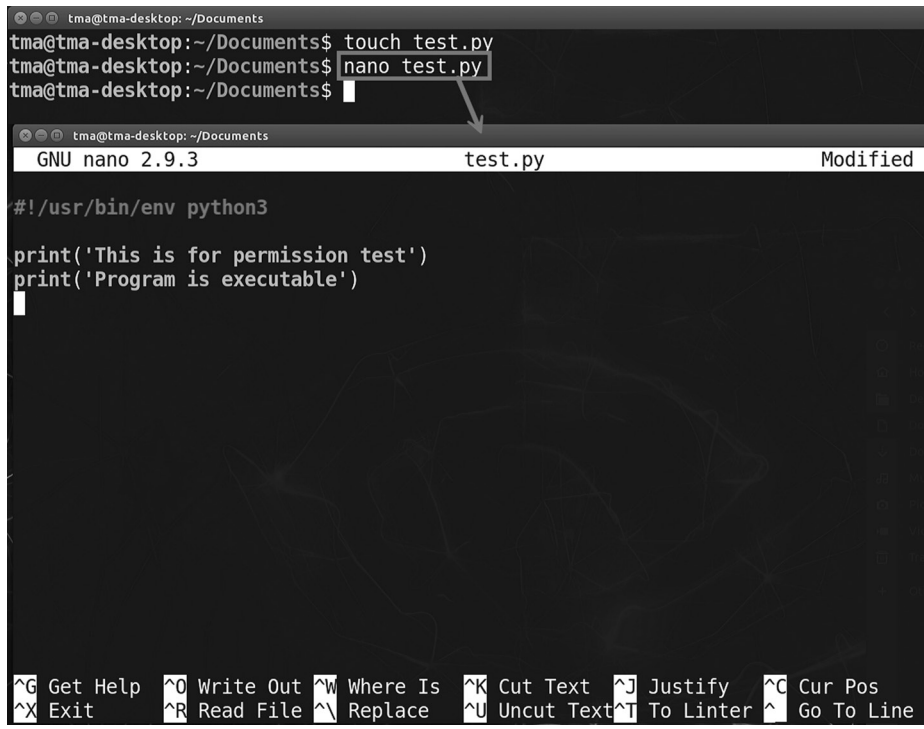


Figure 6.33 Making an executable Python file using Nano editor.

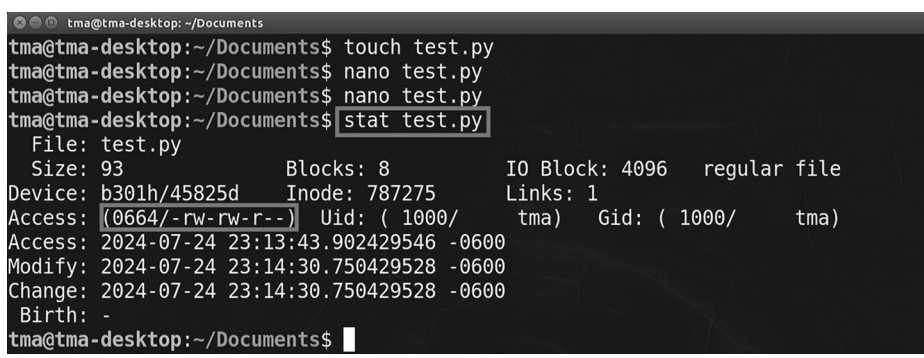


Figure 6.34 Using "stat" command to check the permission access of "test.py" Python file.

in Linux. As shown in Figure 6.34, both numeric and symbolic permissions are displayed. In this example, the user and group have read and write access (rw-), while others have only read access (r--).

At this stage, attempting to run the code directly with "./test.py" will result in an error because the user don't have execute permission for the file. To update this, we can change the permission using the command "chmod +x test.py," which grants execution permission to the user, group, and others. After this, we can successfully run the file using "./test.py" from the terminal. Notice that the new permission has changed to "775" or "rwxrwxr-x," indicating that all users now have execution rights (Figure 6.35).

```
tma@tma-desktop:~/Documents$ ./test.py
bash: ./test.py: Permission denied
tma@tma-desktop:~/Documents$ chmod +x test.py
tma@tma-desktop:~/Documents$ ./test.py
This is for permission test
Program is executable
tma@tma-desktop:~/Documents$ stat test.py
  File: test.py
  Size: 93              Blocks: 8          IO Block: 4096   regular file
Device: b301h/45825d    Inode: 787275      Links: 1
Access: (0775/-rwxrwxr-x)  Uid: ( 1000/     tma)   Gid: ( 1000/     tma)
Access: 2024-07-24 23:16:51.802429475 -0600
Modify: 2024-07-24 23:14:30.750429528 -0600
Change: 2024-07-24 23:16:45.978429477 -0600
 Birth: -
tma@tma-desktop:~/Documents$
```

Figure 6.35 Change permissions of the "test.py" file using the "chmod +x" command to grant execute permission to all users.

```
tma@tma-desktop:~/Documents$ chmod u=r,g=r,o= test.py
tma@tma-desktop:~/Documents$ stat test.py
  File: test.py
  Size: 93              Blocks: 8          IO Block: 4096   regular file
Device: b301h/45825d    Inode: 787275      Links: 1
Access: (0440/-r--r-----)  Uid: ( 1000/     tma)   Gid: ( 1000/     tma)
Access: 2024-07-24 23:16:51.802429475 -0600
Modify: 2024-07-24 23:14:30.750429528 -0600
Change: 2024-07-24 23:18:06.898429446 -0600
 Birth: -
tma@tma-desktop:~/Documents$ ./test.py
bash: ./test.py: Permission denied
tma@tma-desktop:~/Documents$ python3 test.py
This is for permission test
Program is executable
tma@tma-desktop:~/Documents$
```

Figure 6.36 Configure permissions of the "test.py" file to give only read access to the user and group, and no access to others.

Let's examine another example. We can configure permissions so that the user and group have only read access and others have no access by using the set operator (=) (Figure 6.36). In this scenario, the program won't execute directly by itself. However, it can still be run using the Python3 interpreter.

Figure 6.37 illustrates an example where all permissions are set to 0 (a=). In this situation, even the Python interpreter cannot execute the file because it lacks read access.

Figure 6.38 demonstrates another example where read, write, and execute permissions are granted to everyone ("a=rwx"), updating the permission level to "777" or "rwxrwxrwx." At this point, the program is executable by all users.

Using different permission levels in Linux is useful for facilitating teamwork. In collaborative projects on devices such as Jetson Nano, group permissions can be used for shared access and limit unauthorized modifications. Additionally, for embedded systems, where resources are often limited, detailed permission control can be helpful for maintaining operational stability and reliability.

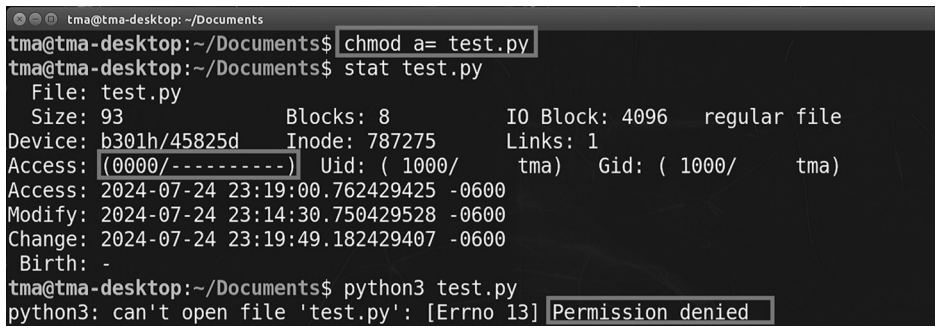


Figure 6.37 Configure permissions of the "test.py" file to no access to all.

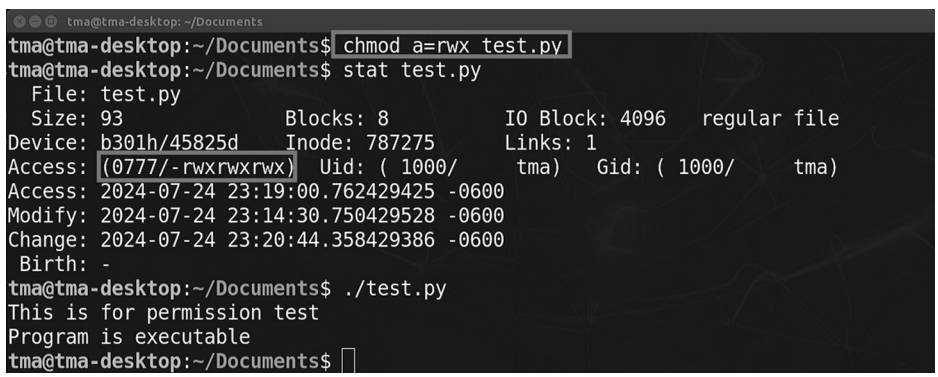


Figure 6.38 Configure permissions of the "test.py" file so that all have read, write, and execute access.

6.10 Install and Uninstall Packages Using "sudo"

In Linux, "sudo" allows authorized users to run commands as the superuser (root) or another user, as defined in the security permission. It is essential for administrative tasks such as software installation, system configuration changes, and user account management. On the Jetson Nano, "sudo" functions similarly to other Linux systems, and it allows users to perform administrative operations. For example, users can update (or refresh) local package lists using "sudo apt-get update" and upgrade all installed packages to the newest versions available in the repositories using "sudo apt-get upgrade" from the terminal. Users can also install Python packages from "PyPI" using "sudo apt-get install python3-pip" and manage user accounts "sudo adduser username."

To remove an installed package "sudo" can be used, but the specific command varies depending on the Linux distribution. In Debian-based distributions such as Ubuntu, the "apt" command is commonly used, and in Red Hat-based distributions such as CentOS and Fedora, "yum" or "dnf" commands are used. Figure 6.39 shows how "sudo" can be used to install and uninstall the Nano text editor from the terminal.

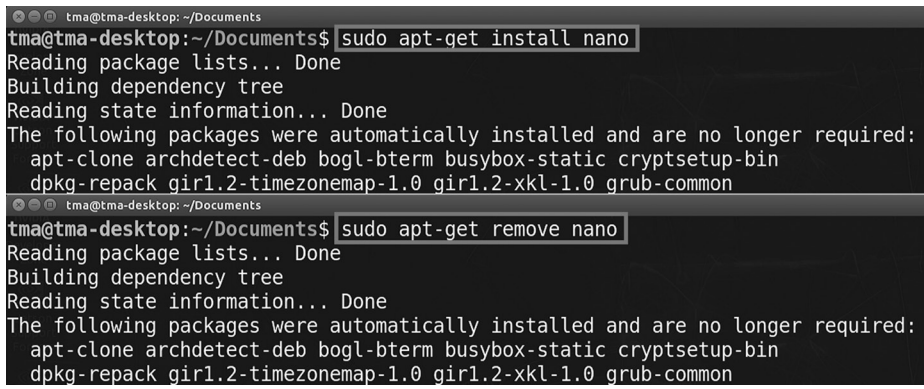


Figure 6.39 Install and uninstall the Nano text editor using "sudo".

6.11 Conclusion

This chapter provides an introductory overview of the Linux terminal and its fundamentals. The command-line operations covered here are crucial for understanding how to perform Python programming tasks from the terminal on Jetson Nano and Raspberry Pi. The chapter includes methods for navigating and managing files and directories, as well as instructions for creating and running Python code. The use of wildcards for efficient file selection is also discussed. Additionally, the chapter addresses finding and viewing files, understanding permissions and ownership, and managing packages with "sudo," all of which are essential for navigating other chapters of this book that use the Linux command terminal.

References

1 Alexandrov, E. (2023). *The Linux Command Line Handbook: An In-Depth Look*, 1e. Amazon Digital Services LLC – Kdp.
2 Thompsons, J. (2017). *Linux: Linux for Beginners Guide to Learn Linux Command Line, Linux Operating System and Linux Commands*. CreateSpace Independent Publishing Platform.
3 Newport, C. (2017). *Linux: The Ultimate Guide to Linux for Beginners, Linux Hacking, Linux Command Line, Linux Operating System, and More*! CreateSpace Independent Publishing Platform.

7

Docker Engine Setup

7.1 Introduction to Docker Engine

The docker engine is a lightweight platform in Linux that enables developers to run applications and their dependencies in a standardized and isolated interface known as containers. It makes sure that applications run smoothly and show consistent performance across different environments. Solomon Hykes and his team at "dotCloud" first designed and showcased the docker system as an open-source project at the PyCon conference [1]. Docker quickly became popular due to its simplicity, efficiency, and the growing need for containerization in various development environments.

A docker system acts like a virtual machine but can utilize a system's kernel features and resources. It is very useful for running deep learning programs as it can ensure reproducibility and make collaboration tasks easier [2]. Deep learning code generally relies on numerous libraries and packages, and minor differences in software versions or configurations can lead to significant variations in performance or even errors. In those scenarios, a docker engine can capture the entire runtime environment with libraries, dependencies, and configurations. Typically, the docker engine can be viewed as a combination of both the docker daemon and the docker client. The docker daemon is a background service running on a host operating system that manages docker containers, and the docker client is the command-line interface tool that users use to communicate with the docker daemon. With the help of commands such as docker run, docker build, or docker pull, the client can send instructions to the docker daemon. Additionally, docker also includes hub (docker hub), which serves as a repository for storing images.

7.2 Docker in Embedded Devices

In embedded devices, docker containers help to simplify the process of setting up and deploying applications in various software environments. Deep learning frameworks in embedded systems often require specific versions of Python, PyTorch, OpenCV, Numpy, CUDA, cuDNN, and other dependencies, and these can be challenging to install by maintaining compatibility. Embedded devices also have relatively low memory and processing power compared to desktop computers. Devices such as the Jetson Nano and Raspberry Pi also run on various Linux distributions, which makes deep learning application deployment very challenging. However, in those resource-constrained embedded devices, we can utilize preconfigured docker images with all the necessary tools and libraries. For example, a docker image can be built with ARM-compatible libraries,

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

especially for the Jetson Nano or Raspberry Pi. This approach ensures that edge computing devices can take full advantage of hardware-specific optimization and run resource-intensive deep learning applications.

In this book, to showcase the use of deep learning on embedded devices, we opted not to use docker engines. Instead, we utilized a pre-built operating system image, which comes with many preconfigured essential packages. However, since docker is widely used during the deployment stage, in this chapter, we have included some common implementations of docker engines designed for Jetson Nano devices.

7.3 Jetson Inference Docker

Since we are already using the Jetson inference library (Chapter 5), we can use the pre-built docker container available in that library. We can pull the docker image directly from NVIDIA's NGC (NVIDIA GPU Cloud) container registry, or we may use the docker from the "jetson-inference" directory. To use the docker, first go to the "jetson-inference" directory using the "cd ~/Downloads/jetson-inference" command. Then check the contents of the directory named docker using "ls docker/" as shown in Figure 7.1. In the list, we find the "run.sh" shell script that can execute a series of commands when the docker container is started. Run this script in the "jetson-inference" directory using the "docker/run.sh" command. It will download the image "dustynv/jetson-inference:r32.6.1," and we would be able to enter commands from the root file system ("/"), which is the top-level directory (starting point of the file system tree) where all other directories reside. The root file system typically contains essential command binaries in "/bin," configuration files in "/etc," user home directories in "/home," user programs and libraries in "/usr," and variable data logs in "/var."

To check the connected video device number (USB camera), enter the command "ls /dev/video*," and it will return the available video capture device number. In Figure 7.2, we see that "video0" device is available. The Jetson inference library provides a tool named "video-viewer" for displaying video streams. Enter "video-view /dev/video0" command to check this display. An example of this application is given in Figure 7.2.

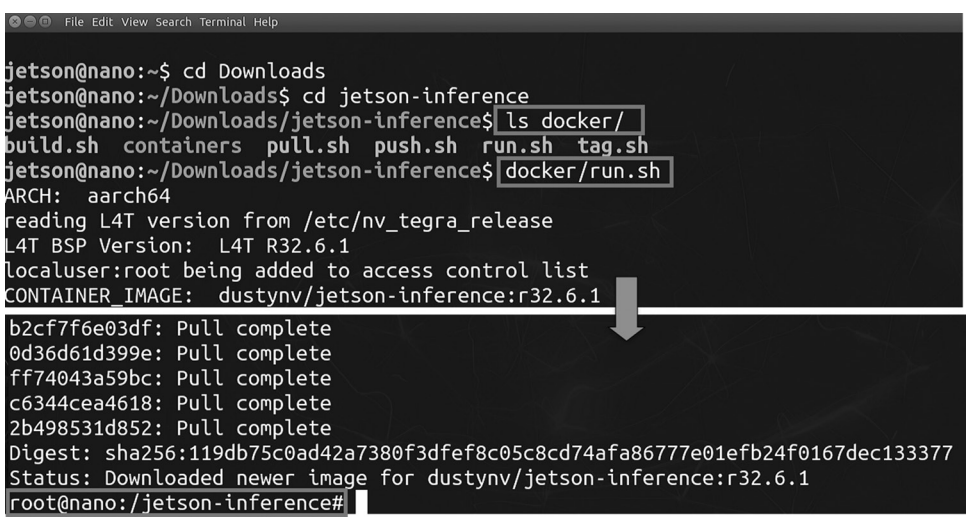


Figure 7.1 Run "run.sh" shell script from the docker to download the latest image for Jetson inference.



Figure 7.2 Use "video-viewer" to check the video feed from device "video0" (USB camera).

Close the window to stop "video-viewer" and enter "ctrl+d" on the keyboard to exit the root file system. In the terminal, we can check the available docker images using "sudo docker images" command, as shown in Figure 7.3. It will show the repository name, tag, and image ID.

7.3.1 Download and Test Pre-Trained Models

Now, we can download pre-trained models from docker and use them for inference. To do this, rerun the docker script from the "jetson-inference" directory and list the files in the root directory using "ls". We'll notice a folder named "tools". Navigate into this folder with "cd tools" and then list its contents using "ls". Inside, we'll find a shell script named "download-models.sh." Run this script by typing "./download-models.sh" from the "tools" directory. This will open the model downloader interface, "Hello AI World (Jetson-inference)" (Figure 7.3). To download the "GoogleNet" model for classification tasks, use the keyboard spacebar to select or deselect models. Deselect any other models by pressing the spacebar. Use the down arrow key to scroll, move the cursor to the "OK" button, and press enter to proceed. This will download the pre-trained GoogleNet model. Note that this operation will need a good internet connection, preferably through an Ethernet cable.

After downloading the model, we can use it for inference. There is a program called "imagenet.py" in the "bin" folder that can facilitate this process. To access it, navigate to the "bin" folder by using the command "cd build/aarch64/bin" from the root directory. This folder contains the compiled binaries for the Jetson Nano's ARM 64-bit architecture (AArch64). We can check the contents of this folder by running "ls", where we will find the "imagenet.py" program. To run the program, use the command "./imagenet.py --network=googlenet /dev/video0." This will start the GoogLeNet model performing classification inference on video frames from the USB camera. The first run might take 2–5 minutes to initialize before the inference begins. After that, the inference will start much faster. To stop the program, close the video streaming window and press "ctrl+d" to exit the root session (Figure 7.4).

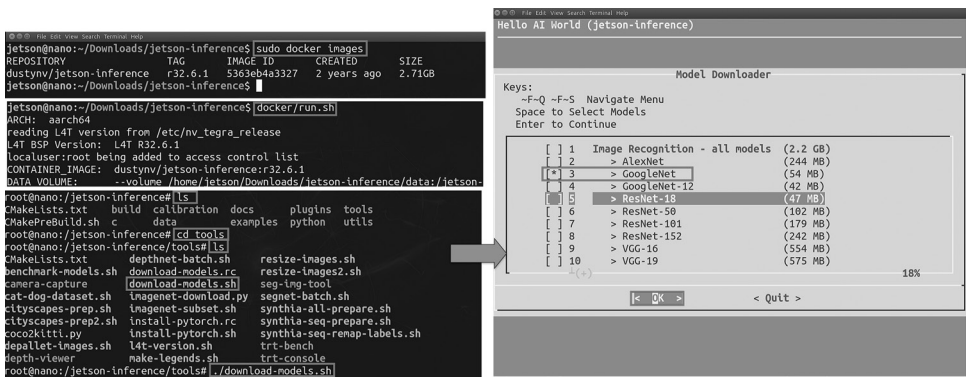


Figure 7.3 Download the pre-trained "GoogleNet" model using the model downloader tool of Jetson inference.



Figure 7.4 Use the pre-trained model "GoogLeNet" from the root for image classification.

7.4 Using Host Files in Docker Environment

In this section, we will show how we can access a Python file saved in a regular host directory from the root environment. For this demonstration, we would like to download another model by repeating the process shown in Figure 7.3. Let's say we want to do inference using "SSD-Mobilenet-v2," which is a deep learning model used for real-time object detection tasks. This model integrates the single shot multibox detector (SSD) with the MobileNetV2 backbone, making it very efficient for applications in embedded systems where computational resources are limited [3, 4]. Download the model using the model downloader, as shown in Figure 7.5. After downloading the model, exit from the root session by pressing "ctrl+d".

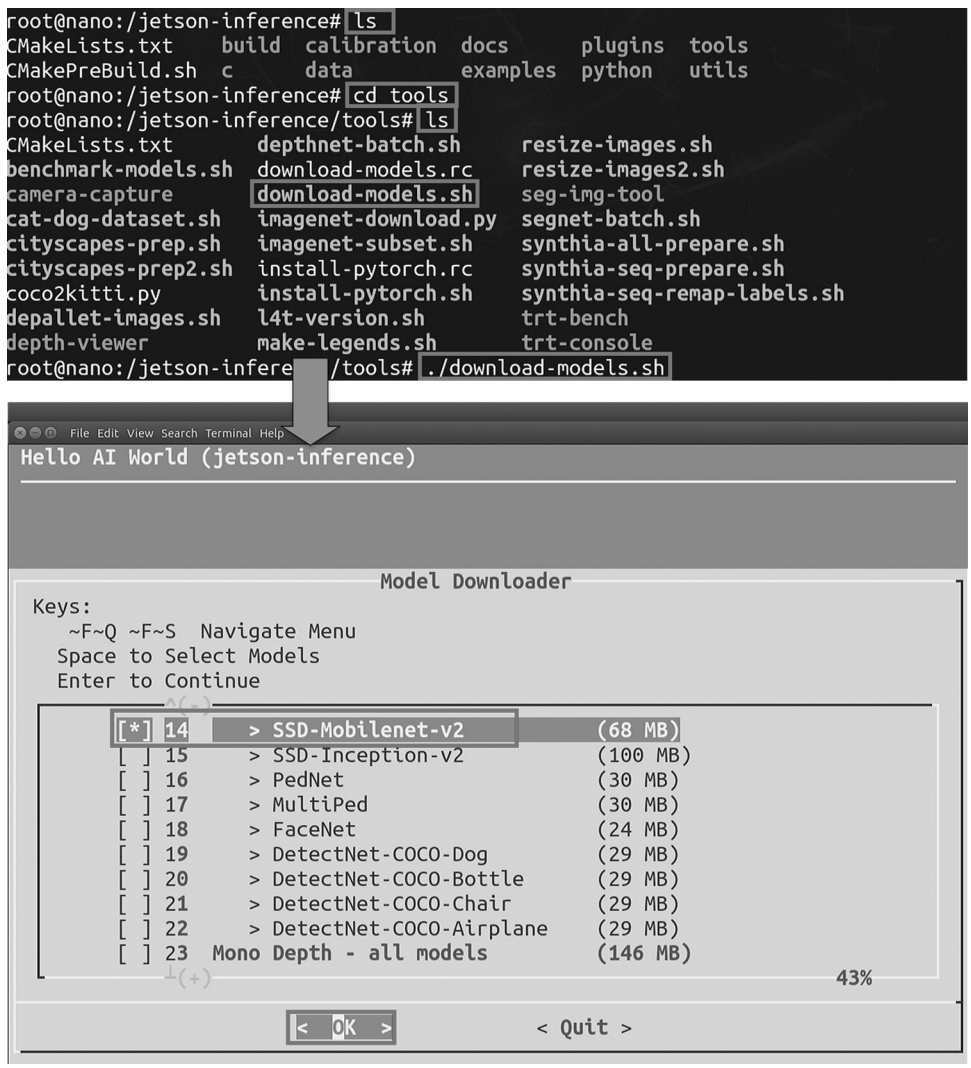


Figure 7.5 Download "SSD-Mobilenet-v2" for the object detection task.

7.4.1 Create Executable Python

To test the inference with a custom Python script, start by creating a directory named "docker-test" inside the "Downloads" folder using the command "mkdir docker-test." Navigate to this directory by running "cd docker-test" and create a file called "example1.py" within the "docker-test" directory using the touch command (Figure 7.6). Next, give execute permission to this file with the command "chmod +x example1.py." To edit its content, open the file in VS Code in the editor.

Figures 7.7 and 7.8 show the "example1.py" program to perform object detection tasks using USB camera input. First, we import "jetson.inference" for running inference, "jetson.utils" for handling utilities such as video streaming, and "argparse" for handling command-line arguments. The "argparse.ArgumentParser" is used in the program to create a parser object, which is then used to define four command-line arguments from the terminal, such as "—camera" for specifying the

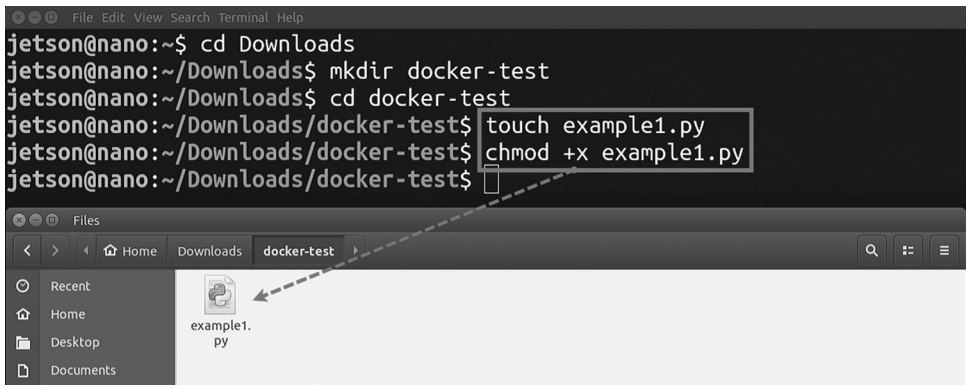


Figure 7.6 Create "docker-test" folder and "example1.py" file inside it, and make the file executable using "chmod."

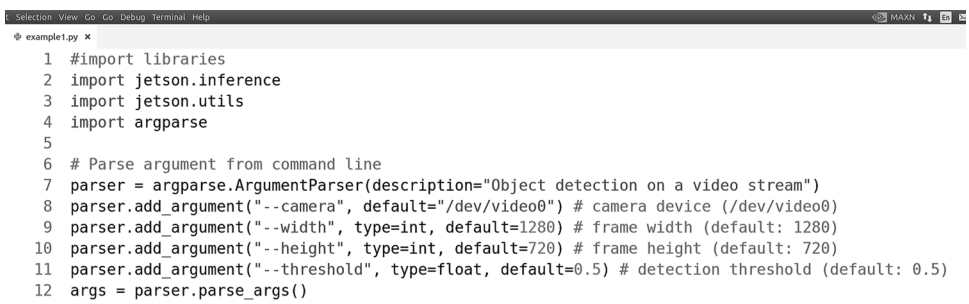


Figure 7.7 Import libraries and parse arguments from the command line interface.

```
14  # Load the detection network ssd-mobilenet-v2
15  net = jetson.inference.detectNet("ssd-mobilenet-v2", threshold=args.threshold)
16  # Create the camera source
17  camera = jetson.utils.videoSource(args.camera,argv=[f"--input-width={args.width}", f"--input-height={args.height}"])
18  # Create the display output
19  display = jetson.utils.videoOutput("display://0")
```

Figure 7.8 Python code to load detection network and capture video for inference.

video input source (e.g. /dev/video0), "—width" and "—height" for setting the resolution of the video frames (default size is 1280 by 720 pixels), and "—threshold" for determining the confidence threshold at which an object is considered detected (defaulting value is set to 0.5). The "args = parser.parse_args()" processes these arguments and makes them accessible in the script for object detection parameters.

For the real-time object detection task, we load the pre-trained SSD-MobileNet-v2 neural network with a specified detection threshold. Then, we initialize the video input source (USB camera) with user-defined width and height parameters and setup a display output for visualizing the results.

Figure 7.9 shows the Python script that performs real-time object detection on the video stream using Jetson inference. It starts an infinite loop that continues as long as the video is streaming. Inside the while loop, a frame is captured from the video stream, and object detection is performed using the pre-trained model. The detection results, which include bounding boxes, class labels, and

```
# Start an infinite loop
while display.IsStreaming():
    # Capture the frame
    img = camera.Capture()

    # Perform object detection on the frame
    detections = net.Detect(img, overlay="box,labels,conf")

    # display detection results
    print(f"Detected {len(detections)} objects:")
    for d in detections:
        print(f"{net.GetClassDesc(d.ClassID)}: {d.Confidence:.3f} at "
              f"({d.Left:.0f}, {d.Top:.0f}, {d.Width:.0f}, {d.Height:.0f})")

    # Display results
    display.Render(img)
    # Update status bar
    display.SetStatus(f"SSD-MobileNet-v2 | FPS: {net.GetNetworkFPS():.0f}")
    # Print profiler times
    net.PrintProfilerTimes()
# Release resources
print("Releasing resources...")
del camera
del display
del net
print("Resources released. Thanks!")
```

Figure 7.9 While loop for real-time object detection on the video stream using Jetson inference.

confidence scores, are printed in the console and overlaid on the video frame for display. The script also updates the display's status bar with the current frame rate and prints information about the model's performance. If the window is closed, the streaming stops and the script releases the resources used by the camera, display, and model. This ensures a clean exit at the end. The script can be executed with a command that specifies the camera device, frame dimensions, and detection threshold.

7.4.2 Attach Host Directory to Root System and Testing

To run the script within the Jetson inference docker environment, the "docker-test" directory needs to be mapped into the docker container. This can be done using the command "docker/run.sh --volume ~/Downloads/docker-test:/docker-test," which links the host's directory with the docker container, allowing files within "~/Downloads/docker-test" on the host system to be accessible in the docker container at "/docker-test." This setup is beneficial for running scripts, testing code, or manipulating files within the container. Once inside the Jetson inference docker environment, we can navigate to the mapped directory by entering "cd /docker-test/." Then, we can run the "example1.py" script with the command "python3 example1.py --camera /dev/video0," which will continuously perform inference on video frames from the USB camera attached to the Jetson Nano. Figure 7.10 shows the commands and object detection results using SSD-MobileNet-v2 on the video stream.

In Figure 7.10, since no arguments are passed for "--width," "--height," and "--threshold," the script will use the default resolution of 1280 × 720 and a detection threshold of 0.5. If needed, we can input these parameters to observe how they affect the inference results. For example, to use a



Figure 7.10 Detection results using SSD-MobileNet-v2 on the video stream using a host file.

frame width of 620 pixels and a height of 480 pixels, with a threshold of 0.7, we can use the command "python3 example1.py –camera /dev/video0 --width 620 --height 480 --threshold 0.7." It will detect objects with a confidence level higher than 70% in a 620 × 480 frame (Figure 7.11).

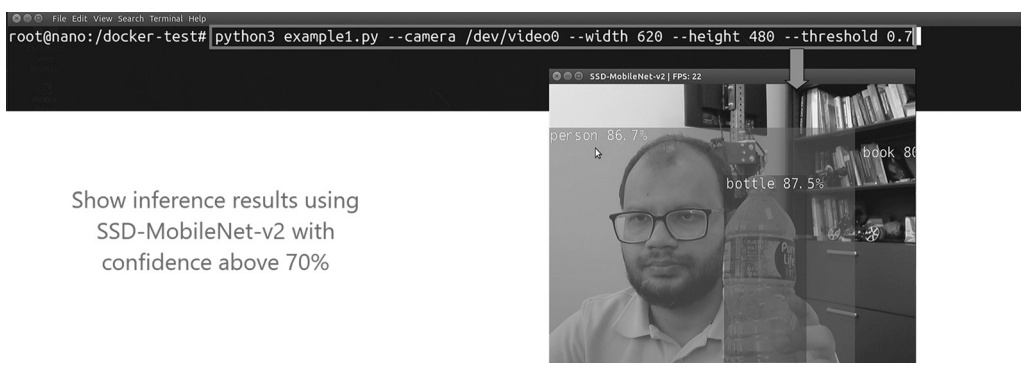


Figure 7.11 Object detection using a 70% threshold in a 620 by 480 pixels frame.

7.5 Building a Docker Image

In this section, we show how to create a docker image containing an executable Python program. This approach can be very helpful in certain situations. The Jetson image we installed in Chapter 5 includes most of the core libraries needed for deep learning and machine learning tasks. However, there are instances where we might require an additional library whose installation could potentially interfere with or conflict with existing packages, such as Numpy, PyTorch, or OpenCV. In those cases, using a docker image to run a specific program allows us to avoid impacting any existing configuration of operating system.

Let's say we want to use MediaPipe 0.10.7 with Python 3.10 for a project. However, our system currently runs Python 3.6, and upgrading the entire system just for a single package isn't practical. The following section shows one of the solutions for using MediaPipe through a docker image.

7.5.1 Runtime Image for MediaPipe Ecosystem

MediaPipe, developed by Google, is a framework that works across different platforms and can be utilized to create multimode machine learning pipelines for analyzing and processing media [5]. It provides a set of customizable tools and solutions for various applications, including real-time face detection, hand tracking, and pose estimation. This framework is optimized to work well on different devices such as desktops, mobile phones, and embedded systems such as the Jetson devices, Raspberry Pi, and Coral devices.

For the Jetson Nano, MediaPipe is particularly useful due to its efficient use of GPU and CPU capabilities, enabling real-time media data processing for applications such as gesture recognition and object detection. A few very useful runtime images with various Python and OpenCV versions are available at [6]. In this example, we would like to use the docker base image "l4t32.7.1-py3.10.11-ocv4.8.0-mp0.10.7," which employs Python 3.10.11, OpenCV 4.8.0, and MediaPipe 0.10.7.

First, we create a Python file named "example2.py" within the "/Downloads/docker-test" directory. The MediaPipe website offers several examples that can be used to test its build or runtime images. For MediaPipe 0.10.x, there's a program available at https://jetson-docs.com/libraries/mediapipe/overview#docker-images that identifies position landmarks on the human body. This program also requires a model called "pose_landmarker_full.task" for implementation. We copy the program and modify it so that, in addition to pose tracking, it can recognize "hands up" and "hands down" gestures. To add this extra functionality, two functions are introduced to the original program, as illustrated in Figure 7.12.

The "check_hands_position" function is created to assess the position of a person's hands relative to their shoulders using pose landmarks detected by MediaPipe's model. It first checks if any pose landmarks are present. If none are found, it returns "Neutral." If landmarks are available, the function retrieves the coordinates of the left and right wrists and shoulders and compares the y-coordinates (vertical positions) of the wrists and shoulders to determine if any of the hands are raised above the shoulder level. If the left or right wrist is positioned above its corresponding shoulder, the function returns "Hands Up." Otherwise, it returns "Hands Down." The "print_result" function was originally in the provided test program. In this function, we added the results found in "check_hands_position" to be printed on the output video frame using OpenCV. The rest of the "example2.py" program is identical to the test program provided at https://jetson-docs.com/libraries/mediapipe/overview#docker-images. The complete "example2.py" program is also available in the chapter resources.

```
def check_hands_position(pose_landmarks):
    if pose_landmarks:
        left_wrist = pose_landmarks[0][mp.solutions.pose.PoseLandmark.LEFT_WRIST]
        right_wrist = pose_landmarks[0][mp.solutions.pose.PoseLandmark.RIGHT_WRIST]
        left_shoulder = pose_landmarks[0][mp.solutions.pose.PoseLandmark.LEFT_SHOULDER]
        right_shoulder = pose_landmarks[0][mp.solutions.pose.PoseLandmark.RIGHT_SHOULDER]

        left_hand_up = left_wrist.y < left_shoulder.y
        right_hand_up = right_wrist.y < right_shoulder.y

        if left_hand_up or right_hand_up:
            return "Hands Up"
        else:
            return "Hands Down"
    return "Neutral"
def print_result(detection_result: vision.PoseLandmarkerResult, output_image: mp.Image,
                 timestamp_ms: int):
    global to_window
    global last_timestamp_ms
    if timestamp_ms < last_timestamp_ms:
        return
    last_timestamp_ms = timestamp_ms

    annotated_image = draw_landmarks_on_image(output_image.numpy_view(), detection_result)

    hand_position = check_hands_position(detection_result.pose_landmarks)
    if hand_position == "Hands Up":
        cv2.putText(annotated_image, 'Hands Up', (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
    elif hand_position == "Hands Down":
        cv2.putText(annotated_image, 'Hands Down', (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

    to_window = cv2.cvtColor(annotated_image, cv2.COLOR_RGB2BGR)
```

Figure 7.12 Add "hands up" and "hands down" recognition functionality using MediaPipe.

7.5.2 Install Docker Extension

In the VS Code editor, search for "dockerfile" in the extensions marketplace and select "install" to install it (Figure 7.13). This will add a docker icon to the left sidebar of VS Code. Ensure that your Jetson Nano is connected to the internet for the installation to proceed.

7.5.3 Create Docker File and Build Image

Next, create a docker file named "dockerfile" without any extension inside the "docker-test" folder. The contents of the docker file are given in Figure 7.14. It sets up an environment for running a Python script using a base image that includes MediaPipe, Python 3.10.11, and OpenCV 4.8.0. This image is specifically built for the Jetson Nano's L4T 32.7.1 platform by MediaPipe developers. The docker file then sets the working directory to "/usr/src" and copies two files into this directory: the modified Python test script "example2.py" and the MediaPipe's model file "pose_landmarker_full.task." Make sure that these two files already exist inside the "docker-test" folder. To run this "example2.py" using Python 3.10 when a container is launched, a "CMD" command is used. This container is designed to perform pose estimation tasks and recognize "Hands Up" or "Hands Down" condition using MediaPipe.

Save both the "dockerfile" and "example2.py" files, then build the image by right-clicking on the "dockerfile" in the editor and selecting "Build Image." A window will appear at the top of the VS Code editor, prompting to assign a name tag to the image file. The default name is "docker-test:latest." We can press enter to accept this name or change it if needed. The first time building an image may take a while as the necessary components are downloaded and the image is constructed. If we didn't expand memory storage in Chapter 5, we might encounter a memory error,

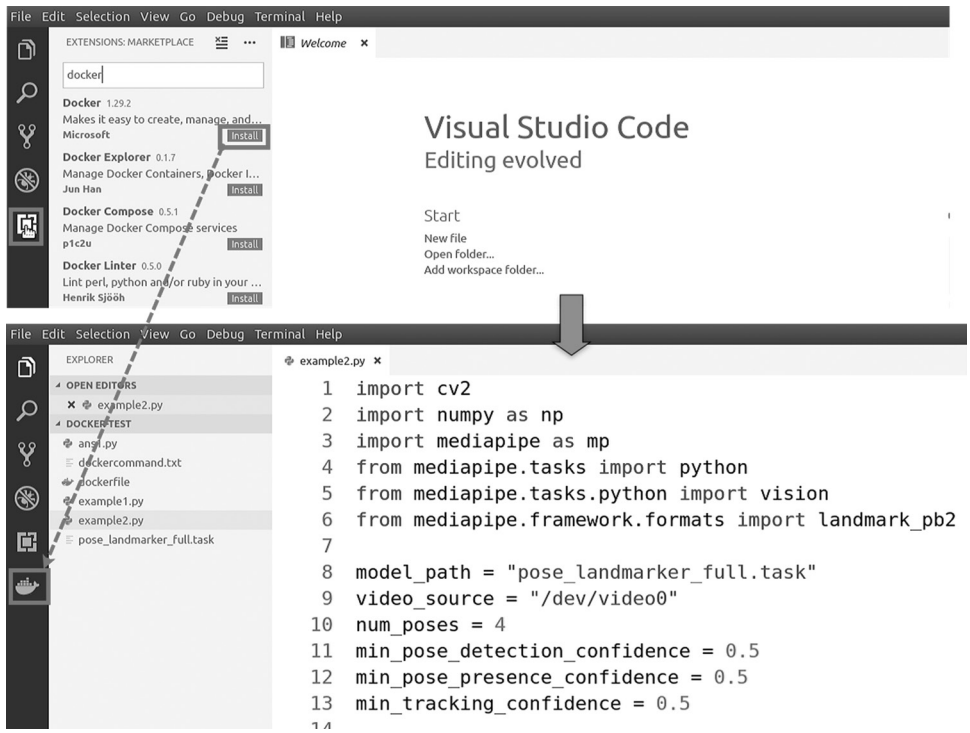


Figure 7.13 Install the docker extension in the VS Code editor.

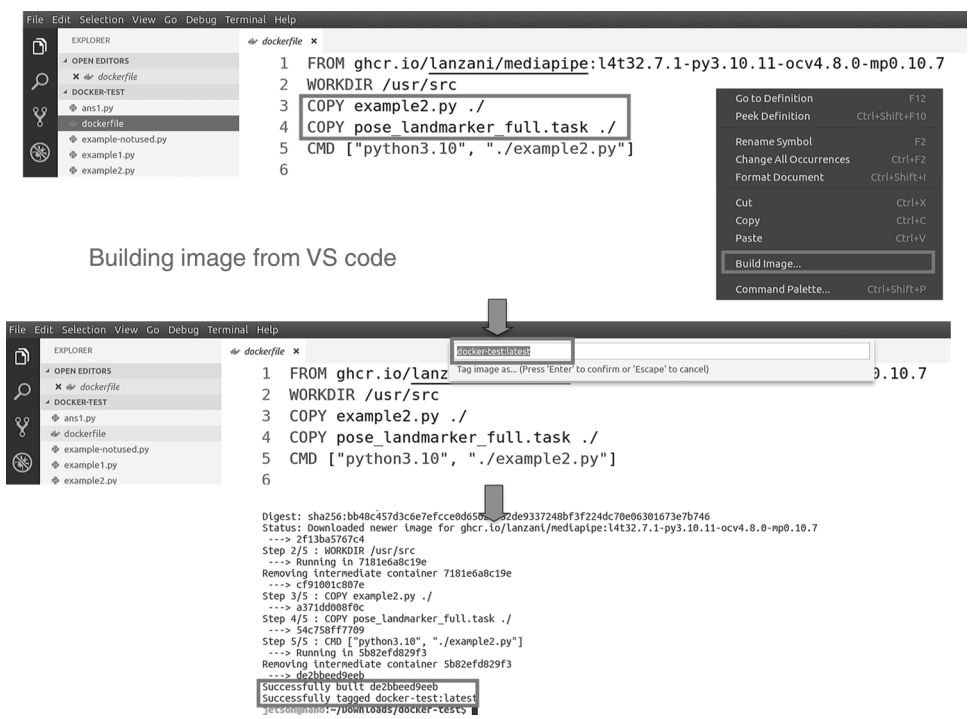


Figure 7.14 Create a docker file and build image to run the "example2.py" script from the container.

preventing the system from building the image. Also, it's recommended to have a stable internet connection, ideally through an Ethernet cable, for the initial build. Once the image is successfully built, the editor's console will display notifications of successful completion, as shown in Figure 7.14.

After building the image, if we check the existing docker images in the terminal using "docker images" we will see the repository "docker-test" is created with the tag "latest."

7.6 Run Python Through Docker Container

To run the Python script "example2.py" from the docker container, we will need to configure the Jetson device so that the user "jetson" can run the docker without entering "sudo" at the beginning. We create a docker group using "sudo groupadd docker," which is used to manage user permissions for docker. The user "jetson" is then added to this group using "sudo usermod -aG docker jetson," enabling the user to execute docker commands without superuser privileges. The permissions of the docker socket file ("/var/run/docker.sock") also needed to be changed as reading and writing by all users "sudo chmod 666 /var/run/docker.sock." This will allow any user on the system to communicate with the docker daemon. Finally, the docker service is restarted to apply these changes using "sudo systemctl restart docker." These commands need to be entered in the Jetson terminal as given below:

$ sudo groupadd docker
$ sudo usermod -aG docker jetson
$ sudo chmod 666 /var/run/docker.sock
$ sudo systemctl restart docker

This setup will ensure that the user "jetson" can now use docker commands without needing to invoke "sudo." If we use a different user name, replace the name "jetson" after "sudo groupadd" with the new user name.

Next, use "export DISPLAY=:0" in the terminal to set the DISPLAY environment variable for the primary display. Use "xhost +" to allow any user to connect to the X server, which is necessary if we use graphical applications inside the container. In Figure 7.15, we run the code multiple times, which is why these commands are not shown. But for the first run, enter the following in the terminal:

$ export DISPLAY=:0
$ xhost +

Then to run the script use "docker run -dit --gpus all --privileged -e DISPLAY=$DISPLAY -v /tmp/:/tmp/ --network=host --name=cc -it docker-test:latest" as shown in Figure 7.15. This command initiates the docker container with comprehensive GPU access, elevated privileges, and host network integration. The container is also configured to utilize the host's display, share the /tmp directory, and run in detached mode while maintaining interactivity.

The name is assigned as "cc" and based on the "docker-test:latest" image, but it needs to be updated (e.g. xx, xxx, xxxx or c, cc, cccc) every time we run the container.

Figure 7.15 shows that the Python script "example2.py" is running from the container even though we don't have Python 3.10 and MediaPipe installed on the system.

If you are dealing with multiple docker images, it is better to clean up any unnecessary repository after some time to save disk space. To delete a docker image, first check the docker image ID using

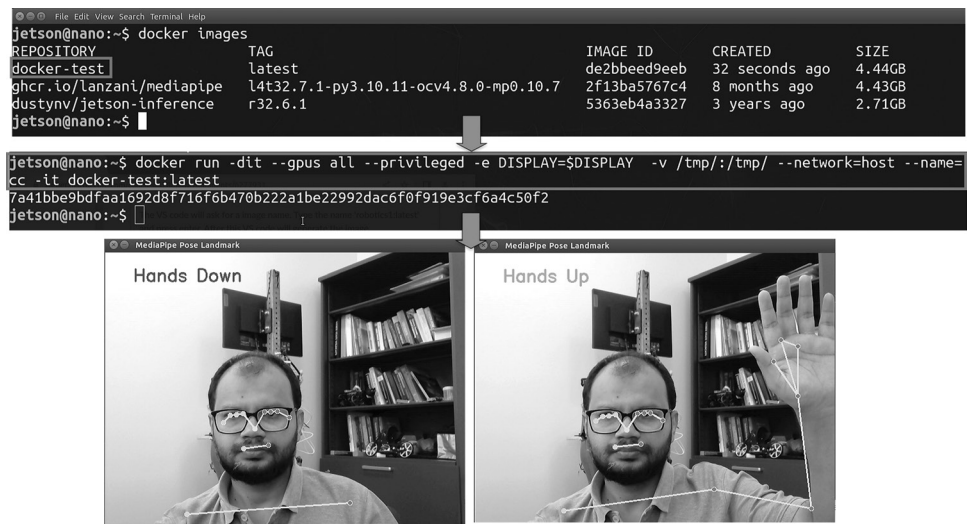


Figure 7.15 Running "example2.py" from the "docker-test" to recognize between hands up and hands down positions.



Figure 7.16 Delete a docker image using image ID from the terminal to save disk space.

"docker images" from the terminal and then force delete the container using "docker rmi -f ImageID," as shown in Figure 7.16.

7.7 Exercise Problems

(Q1) Jetson inference library offers several pre-trained models that can be downloaded using the model downloader tool. Among these models is "pose-resnet18-hand," a deep learning model capable of detecting and tracking human finger poses, similar to the functionality provided by the MediaPipe library. Download this model (using download_models.sh from the Jetson inference's docker) and develop an inference program to analyze video frames for finger tracking, as shown in Figure 7.17. Make this program executable using "chmod +x programname.py" and run it from the root directory. Some hints for the code are given in Figure 7.18.

(Q2) Build a docker image to run a script using Python 3.8, OpenCV 4.8.0, and MediaPipe 0.10.7. Use a test Python program to import these libraries and show versions in the output.



Figure 7.17 Download and use the "pose-resnet18-hand" model to track fingers using Jetson-inference.

```
# Parse argument from command line
parser = argparse.ArgumentParser(description="Hand pose estimation on a video stream")
parser.add_argument("--camera", default="/dev/video0") # camera device (/dev/video0)
parser.add_argument("--width", type=int, default=1280) # frame width (default: 1280)
parser.add_argument("--height", type=int, default=720) # frame height (default: 720)
parser.add_argument("--threshold", type=float, default=0.1) # keypoint threshold (default: 0.1)
args = parser.parse_args()

# Load the pose estimation network pose-resnet18-hand
net = jetson.inference.poseNet("pose-resnet18-hand", threshold=args.threshold)

# Create the camera source
camera = jetson.utils.videoSource(args.camera, argv=[f"--input-width={args.width}", f"--input-heig
```

Inside while loop:

```
    # Perform hand pose estimation on the frame
    poses = net.Process(img, overlay="links,keypoints")

    # Display detection results
    print(f"Detected {len(poses)} hands:")
    for pose in poses:
        print(f"Hand {pose.ID}")
        for i in range(len(pose.Keypoints)):
            keypoint = pose.Keypoints[i]
            print(f"  Keypoint {i}: ({keypoint.x:.0f}, {keypoint.y:.0f})")

    # Display results
    display.Render(img)
```

Figure 7.18 Modifications inside while loop for tracking fingers.

References

1. Johnston, S. (2021). *Docker: Nine Years YOUNG*. https://www.docker.com/blog/docker-nine-years-young/.
2. Cook, J. (2017). The docker engine. *Docker for Data Science*, 71–79. USA: Apress L. P.
3. Liu, W., Anguelov, D., Erhan, D. et al. (2016). SSD: single shot multibox detector. *arXiv*: 1512.02325. Computer Vision – ECCV 2016. ECCV 2016. Lecture Notes in Computer Science, volume 9905. Springer, Cham. https://doi.org/10.1007/978-3-319-46448-0_2.
4. Sandler, M., Howard, A.G., Zhu, M. et al. (2018). MobileNetV2: inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* 4510–4520.
5. Lugaresi, C., Tang, J., Nash, H. et al. (2019). MediaPipe: a framework for building perception pipelines. *arXiv*: abs/1906.08172.
6. Mediapipe (2023). *Docker Images*. https://jetson-docs.com/libraries/mediapipe/overview#docker-images.

8

Dataset Development

8.1 Introduction and Requirements

A collection of labeled data is very critical in deep learning, as it is used to train, validate, and test a deep learning model. While labeled data is primarily used for training, sometimes unlabeled data is also utilized. The quality and diversity of a dataset directly influence the model's ability to learn and generalize from the data. A high-quality dataset should be diverse, accurately represent the problem domain, and be free from biases or errors. For example, an image dataset for classification should cover various data variations, such as different view angles, lighting conditions, and noise levels. These variations will prevent overfitting, where the model performs well on training data but poorly on new examples, and underfitting, where the model fails to capture essential patterns in the data. Therefore, building a robust dataset is critical for creating effective deep learning models that are applicable to real-world scenarios. In this chapter, we show how to create datasets for some common deep learning tasks such as image classification, object detection, image segmentation, and image captioning.

8.2 Types of Datasets

In deep learning, various types of datasets can be used depending on the problem being addressed. The datasets can be divided into three main categories based on available label information: supervised, unsupervised, and semi-supervised [1]. Supervised datasets contain labeled input–output pairs for training deep learning or machine learning models. Unsupervised datasets don't have explicit labels and are typically used for clustering or dimensionality reduction. Semi-supervised datasets have a small portion of labeled data with a larger set of unlabeled data. The most common datasets in deep learning are image datasets that can be used for image classification, object detection, and image segmentation tasks. Text datasets are used in natural language processing tasks for sentiment analysis, language translation, and text generation. Audio datasets contain sound recordings that can be used for speech recognition, sound classification, music generation, etc. Video datasets are used for video classification, action recognition, and video captioning, and time series datasets are used for forecasting and anomaly detection. Besides these common types of datasets, reinforcement learning datasets are used for training agents in decision-making scenarios, and synthetic (artificially generated) datasets are used to simulate real-world scenarios [2, 3]. The selection of dataset types and the effectiveness of deep learning models are dependent on available resources for collecting data and the desired output.

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

8.3 Manual Dataset Creation

In this section, we will demonstrate how to create a dataset manually for image classification and object detection. To create a well-labeled dataset, the first step is to define the problem clearly and identify the data requirements, including the classes or categories needed for classification tasks.

8.3.1 Classification and Detection Dataset

Let's assume we want to build an image dataset containing four distinct classes: pencil, scissors, screwdriver, and wrench. The simplest approach is to capture images using a camera or cellphone with adequate resolution. Ideally, natural lighting works best, but it's beneficial to introduce variations, such as different lighting conditions, orientations, and backgrounds. Additionally, photographing objects partially (e.g. a full pencil versus a partially covered pencil) can be advantageous. Generally, the more data we collect, the more robust the model's learning will be. It's also important to ensure that each class has a sufficient number of images.

After taking enough pictures (e.g. 100–200 for each class), we can organize them into different folders using a consistent naming system. In the dataset provided for Chapter 9, "images" inside the "classification1/dataset1" folder are named 0001.jpg, 0002.jpg, 0003.jpg, and so on., and another folder inside "dataset1," "annotations," has annotated ".xml" files that have additional details for these images. Initially, we created a ".csv" file to track the image numbers and their corresponding labels. For example, images 0001.jpg to 0100.jpg are labeled as pencils, while images 0101.jpg to 0200.jpg are labeled as scissors. A screenshot of the ".csv" file is given in Figure 8.1.

	A	B	C	D
1	filename	label		
2	0001.jpg	pencil		
3	0002.jpg	pencil		
4	0003.jpg	pencil		
5	0004.jpg	pencil		
6	0005.jpg	pencil		
7	0006.jpg	pencil		
8	0007.jpg	pencil		
9	0008.jpg	pencil		
10	0009.jpg	pencil		
11	0010.jpg	pencil		
12	0011.jpg	pencil		
13	0012.jpg	pencil		
14	0013.jpg	pencil		

Figure 8.1 Image names and labels for reference in a ".csv" file.

With these images and labels, we can train a deep learning model. However, to improve efficiency, we have drawn bounding boxes around the objects of interest, enabling a more effective model training process. Section 8.3.2 explains how to create ".xml" files for each image with bounding box information.

8.3.2 LabelImg Setup

LabelImg is an open-source graphical image annotation tool used for creating labeled datasets for deep learning projects, particularly in computer vision tasks such as object detection [4]. It allows users to create bounding boxes around objects and assign labels using an intuitive interface. The annotated images in Labelimg can be saved in formats such as XML (Pascal VOC) or TXT (YOLO) and can be exported in other formats compatible with various deep learning frameworks. By streamlining the annotation process, Labelimg enables researchers and developers to efficiently create high-quality labeled and diverse datasets that are suitable for image classification, object detection, and segmentation tasks. To install the LabelImg tool, begin by downloading the ".zip" file from its GitHub repository: https://github.com/HumanSignal/labelImg. Once downloaded, the file named "labelImg-master.zip" should be extracted using an application such as ".7z" or a similar tool. Figure 8.2 shows that the extracted files are located in the "Downloads" folder, where we can find all the "labelImg" files and codes.

Next, we need to install the "labelImg" software within the Anaconda environment. To begin, open the Anaconda Prompt with administrator privileges and navigate to the extracted "labelImg-master" directory. The prompt will be opened from the "C:\Windows\system32" directory, and we can access other locations by using the "cd" command followed by the directory path. In this example, since the

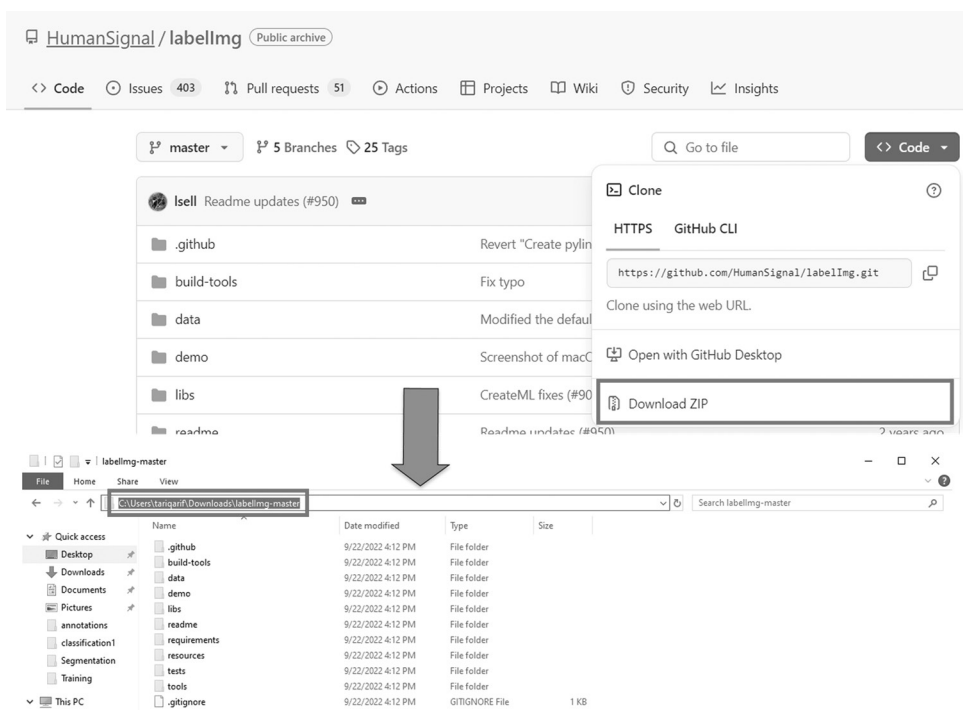


Figure 8.2 Download "labelImg-master.zip" from the GitHub page and extract it.

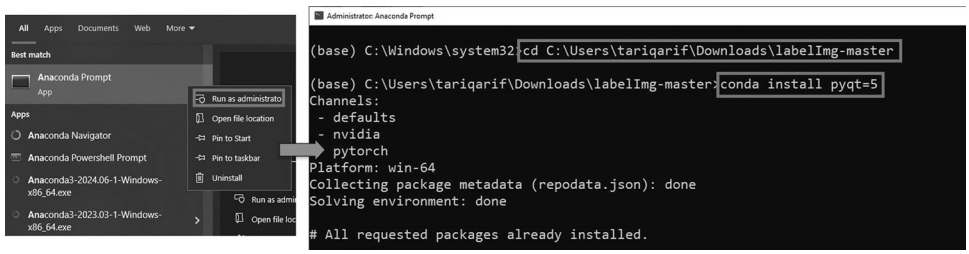


Figure 8.3 Launch Anaconda Prompt as administrator, access the "labelImag-master" directory, and install "pyqt" Python bindings for the Qt application framework.

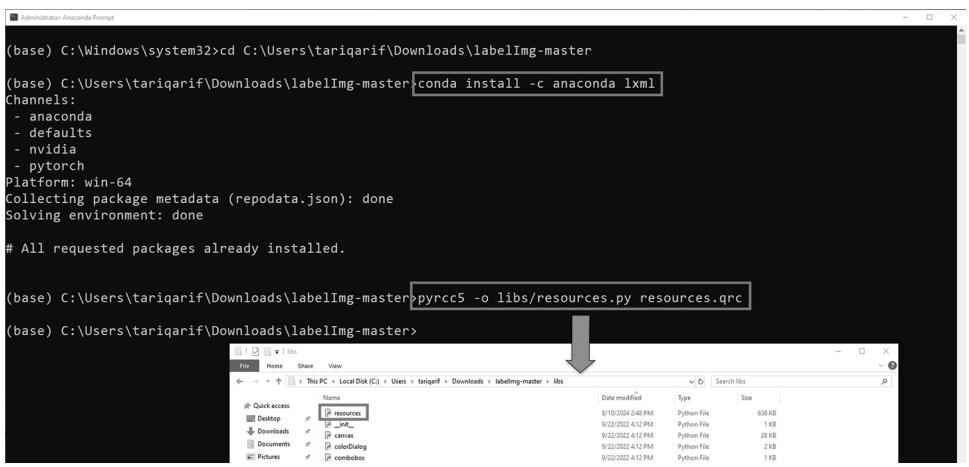


Figure 8.4 Convert the Qt resource file (.qrc) from the "labelImg" folder into a Python module (.py) using "pyrcc5" and save the output (resources.py) file in the "libs" directory.

folder was extracted into the user's "Downloads" folder, we navigate to that location (Figure 8.3). Inside the "labelImag-master" folder, execute the command for installing "pyqt" Python bindings using "conda install pyqt=5." The "pyqt" is a Qt application framework that enables Python developers to create cross-platform graphical user interface programming.

To work with "xml" and "HTML" documents in "labelImg," we will need to install the Python "lxml" library. This can be done by executing "conda install -c anaconda lxml" from Anaconda Prompt. After that, convert the Qt resource file (.qrc) from the "labelImg" folder into a Python module (.py) using "pyrcc5" and save the output (resources.py) file in the "libs" directory (Figure 8.4).

Place the "resources.qrc" file into the "libs" folder, and then start the "labelImg" application by running the command "python labelimg.py" in the Anaconda Prompt. Once the "labelImg" application is open, we'll notice that the dropdown box for labels includes several available classes, as illustrated in Figure 8.5.

The predefined classes in the "labelImg" tool can be customized by editing the "predefined_classes.txt" file located in the "data" directory. For example, for Chapter 9, we created a "dataset1" containing four image classes: pencil, scissors, screwdriver, and wrench. To annotate the images in "dataset1," we need to update the "predefined_classes.txt" file by deleting other names and adding these class names. After making these changes, when we launch "labelImg," the label dropdown on the right side will display the new classes for this dataset (Figure 8.6).

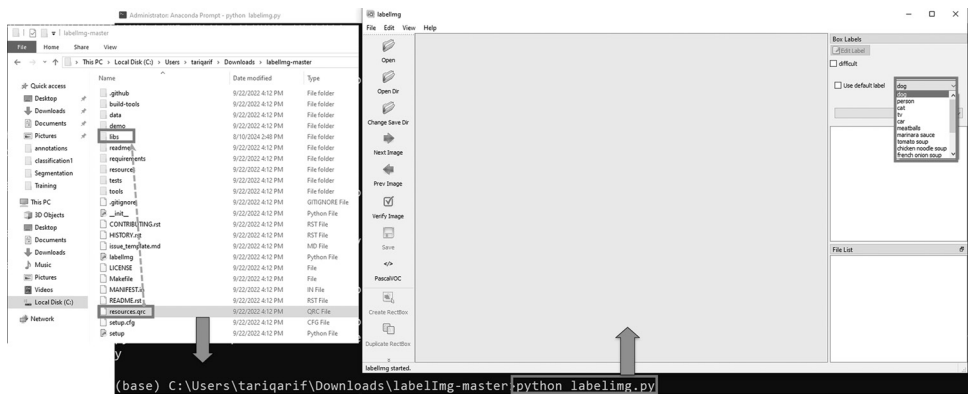


Figure 8.5 Place the "resources.qrc" file into the "libs" folder, and start the "labelImg" application by executing "python labelimg.py" from Anaconda's command prompt.

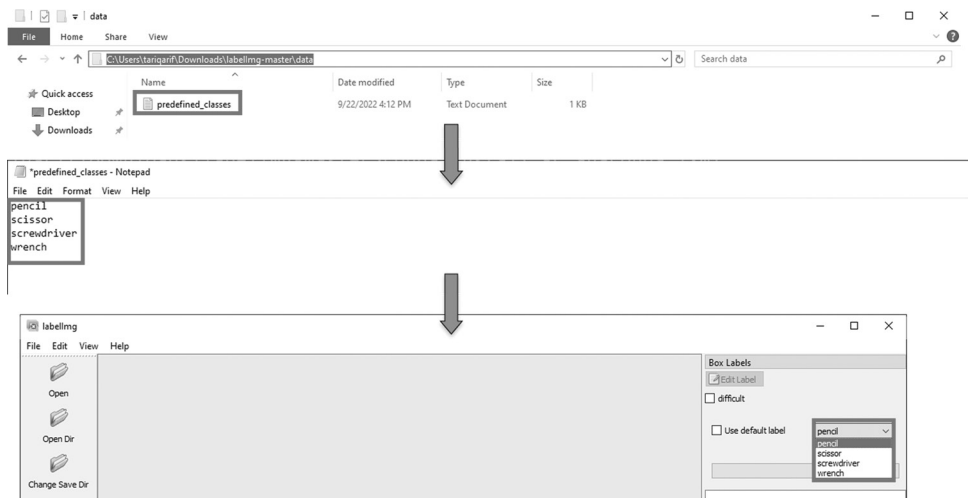


Figure 8.6 Edit the "predefined_classes.txt" file located in the "data" folder to include the new class names for a customized dataset.

To label images, start by opening the directory containing all the images using the "labelImg" tool. Once the directory is open, we'll see a list of all the image files in that directory in the "File list" (located in the lower right corner). To create bounding boxes, click on "Create RectBox" and use the mouse to draw a rectangle around the object. The process is simple and user-friendly (Figure 8.7).

If the images are grouped by similar classes (for example, in Figure 8.1, images 0001.jpg to 0100.jpg are all pencils), we can enable the "Use default label" option for "pencil." This will automatically label all elements within the rectangle as "pencil," speeding up the dataset creation process. When we move on to a different class of images, we can simply update the default label and continue labeling.

After creating the rectangular box (if the default label is not used), a prompt drop-down with class names will appear, allowing us to select the appropriate class (Figure 8.8). Select the class from the menu, then click on "Next Image." A dialogue box will pop up, asking if you want to save

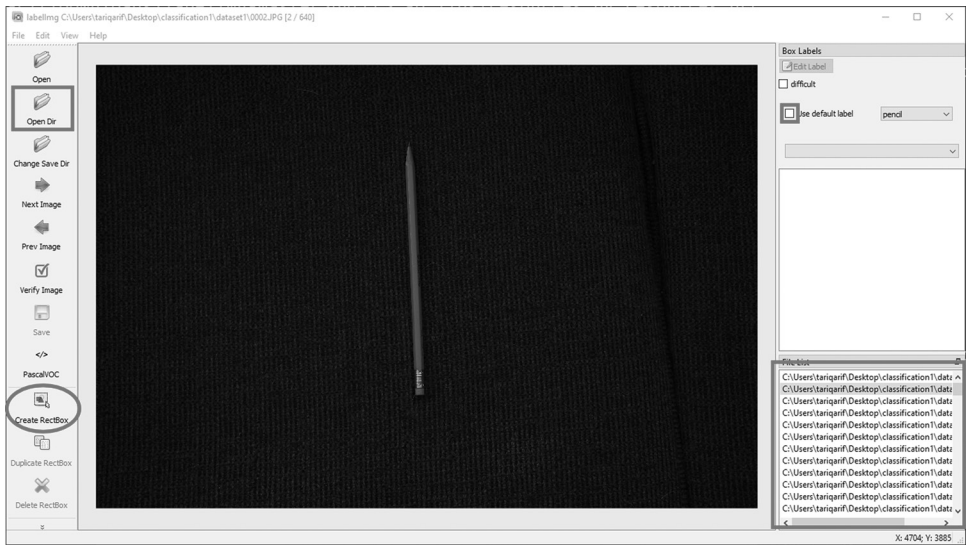


Figure 8.7 Open the directory containing dataset images, select the default label (if similar images are put together), and use "Create RectBox" to create bounding boxes.

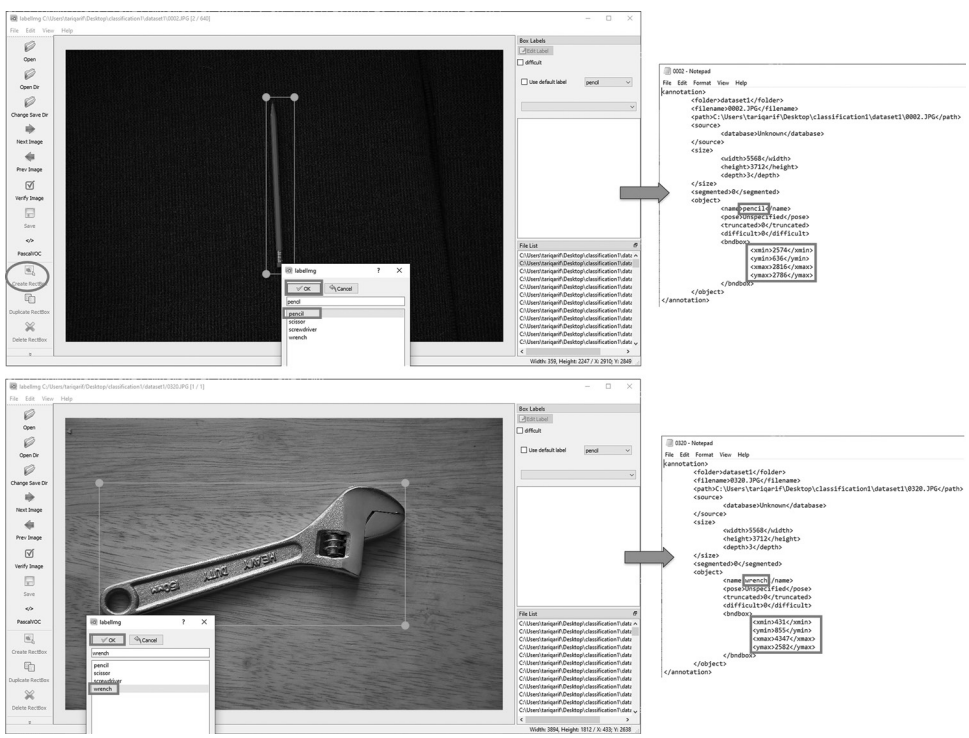


Figure 8.8 Draw a rectangle around the object and choose a class for it. Then, by clicking the "Next Image" arrow and selecting "Ok," "labelImg" will save the corresponding ".xml" file in the image folder.

the corresponding ".xml" file. By clicking "Ok," "labelImg" will save a ".xml" file with the same name as the image (e.g. 0320.xml for 0320.jpg) inside the image folder. This ".xml" file will contain the selected class and the bounding box coordinates, which can be used to train deep learning networks effectively. If there are multiple classes within the same image, we can create multiple rectangles and assign different classes.

8.4 Automatic Image Collection Using Embedded Device

The automatic data collection process using embedded devices is crucial because it allows the fast and efficient creation of high-quality datasets. Moreover, real-time data acquisition offers immediate insights into the quality and structure of the data as it is being collected. In many cases, gathering large amounts of accurate and timely data is challenging and labor-intensive, and therefore, automation is necessary. Embedded devices such as the Jetson Nano and Raspberry Pi can operate continuously and autonomously. So, if automation can be utilized, it can reduce the need for human intervention and minimize errors associated with manual data collection. These embedded devices are also typically low-powered and compact, making them ideal for use in remote or inaccessible locations. In this section, we present a Python program that can be utilized with both the Jetson Nano and Raspberry Pi for an automatic image collection process.

In the Jetson Nano, using Visual Studio Code-oss, a Python file named "capture_data.py" was created. The file is located in the "Home/Desktop/chap-jetson" directory on our system. First, we imported the OpenCV and OS libraries. The "os" library is necessary for interacting with the operating system to create files or directories. Next, the "video_capture" variable is used to initialize the USB camera and capture frames from device number 0. A directory named "captured_images" was also created inside the "chap-jetson" directory to store all the images in chronological order. The "frame_interval" variable defines the frequency of saved frames, which is currently set to 50, i.e. one frame is saved in every 50 frames and stored in the "captured_images" folder. The "image_counter" variable tracks the number of frames, and the "image_number" is used to assign .jpg image names to the saved images. These codes are shown in Figure 8.9.

capture_data.py

```
1  #import libraries
2  import cv2
3  import os
4  # Initialize the usb camera
5  video_capture = cv2.VideoCapture(0)
6
7  # Directory to save the captured images
8  output_dir = 'captured_images'
9  # take picture in every 50 frames
10 frame_interval = 50
11 # image count and number of captured images
12 image_counter = 0
13 image_number = 1
```

Figure 8.9 Python code in Code-oss for importing libraries, initializing USB camera, and defining output directory name, frame interval, image counter, and image numbers.

capture_data.py

```
14
15  while True:
16      # Capture a frame
17      ret, frame = video_capture.read()
18      if image_counter % frame_interval == 0:
19          # Generate a filename
20          image_filename = os.path.join(output_dir, f'image_{image_number}.jpg')
21          # Save the frame as an image
22          cv2.imwrite(image_filename, frame)
23          print(f'Captured {image_filename}')
24          #increase the image number by 1
25          image_number+=1
26      cv2.imshow('Frame', frame)
27      # Increase the image counter by 1
28      image_counter += 1
29      # Hit 'q' on the keyboard to quit (while video window is active)
30      if cv2.waitKey(1) & 0xFF == ord('q'):
31          break
32  # Release handle and destroy windows
33  video_capture.release()
34  cv2.destroyAllWindows()
```

Figure 8.10 "while" loop for continuously capturing and saving frames in the "captured_image" folder.

Next, we use a continuous "while" loop to capture frames from the camera. An if statement is used to create a logic that captures one out of every 50 frames and generates an image file from it. The "cv2.imwrite()" function saves these image files inside the "captured_images" folder using the specified image name and numbers. The loop will break if the "q" key is pressed while the video window is active. Finally, the ".release()" method is used to release the video capture device, and "cv2.destroyAllWindows()" closes any windows that were opened by OpenCV during the script's execution (Figure 8.10).

To run the code, right-click on the VS Code editor page and select "Run Python File in Terminal." A video window will display the current frames, and as frames are saved every 50 frames, the VS Code terminal will continuously show the names of the saved files. If we want to capture images at a slower rate, we can increase the "frame_interval" variable. After closing the program by pressing "q," we can verify the saved images by checking the "captured_images" folder, as shown in Figure 8.11. This Python code does not create separate folders for training or validation image data. This step can be done manually, or we can select a percentage of the total dataset as validation data during the training process in a random manner.

8.5 Automatic Data Labeling

Most of the time, after collecting images, it's necessary to compile all the image names and their corresponding labels into a ".csv" file. This file is useful for fetching training and validation images during the training process. Figure 8.12 shows a sample Python code (csv_file_creation.py) that can be used to label images. The dataset referenced here is "dataset1," which is used in Chapter 9 to demonstrate an image classification problem. Initially, images are stored in a folder with sequential numerical names, such as 0001.jpg, 0002.jpg, 0003.jpg, etc. The Python code illustrates how to label a range of image numbers with specific labels (as shown in the second column of the

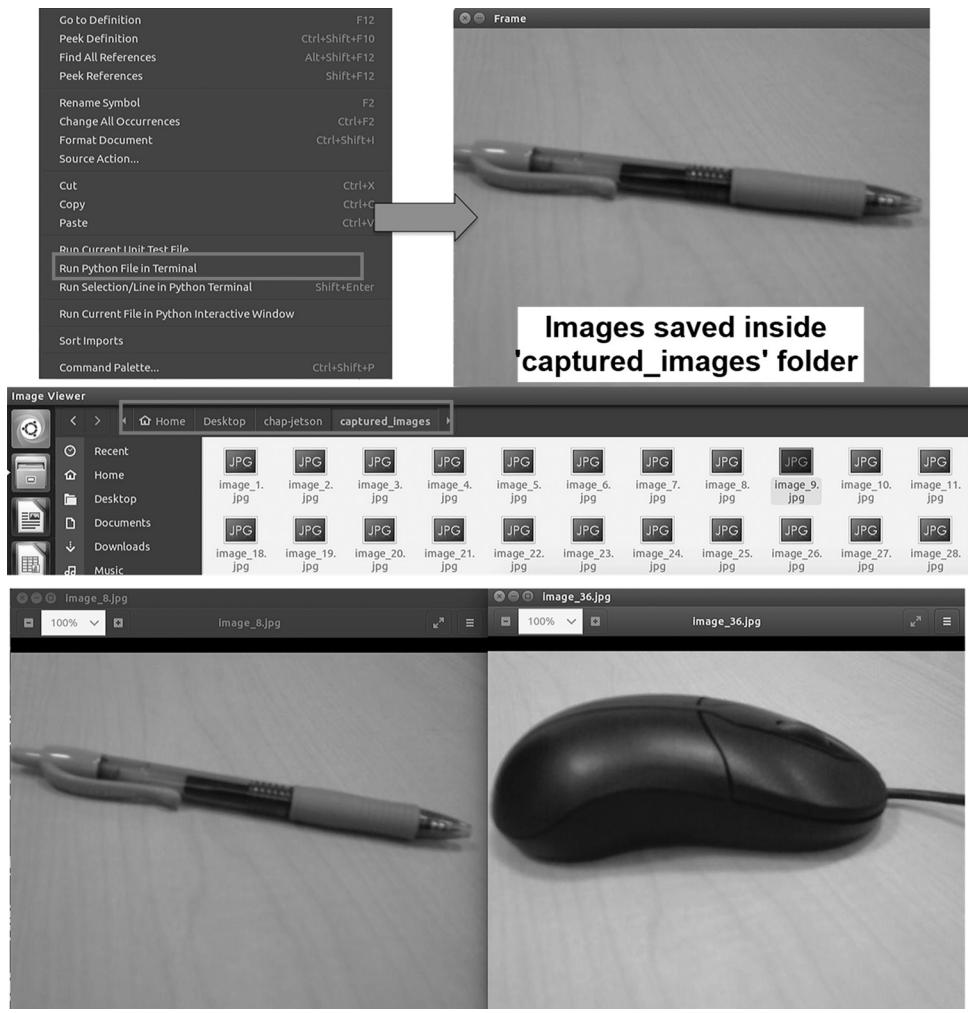


Figure 8.11 Run the Python file in the VS code terminal and save images inside the "capture_images" folder.

CSV file). For example, in this dataset, images numbered from 0581.jpg to 0640.jpg are labeled as "wrench" images. In the Python code, we set the "num_images" variable to 60 and created a CSV file named "image_labels1.csv" with two column headers: "filename" and "label." The filenames column generates file names from 0581.jpg to 0640.jpg, and the corresponding labels (wrench) are assigned in the "label" column. For "dataset1," we created several similar CSV files, then combined all data (0001.jpg to 0640.jpg) into a single CSV, which was later used in deep learning training (Chapter 9) to assign appropriate labels to the program.

Now, let's assume we want to create a CSV file for the "captured_images" shown in Figure 8.11. These images are named "image_1.jpg," "image_2.jpg," "image_3.jpg," and so on. If we wish to label images from "image_21.jpg" to "image_60.jpg" as "mouse," we can modify the Python code. In this updated code shown in Figure 8.13, "num_images" is set to 40, and the "image_name" and "label variables" inside the for loop are adjusted to match with the new naming convention.

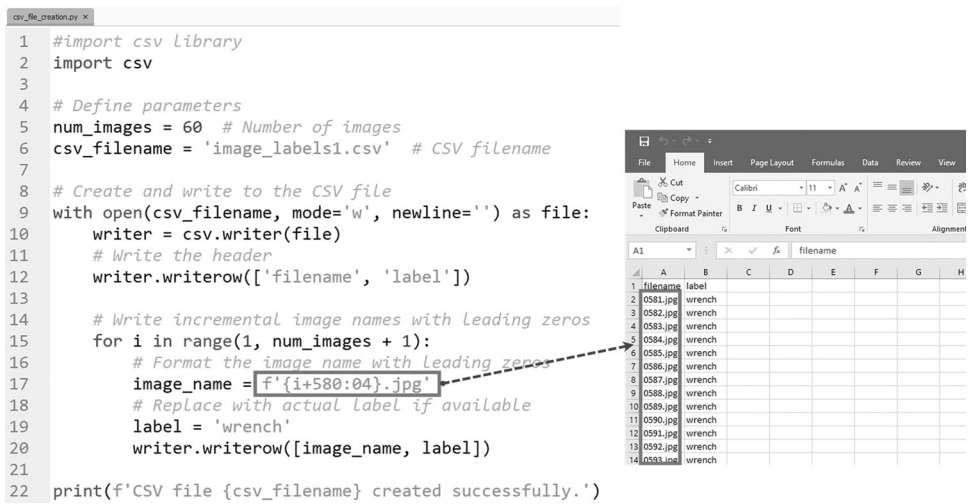


Figure 8.12 Automatic data labeling in CSV files for a range of images.

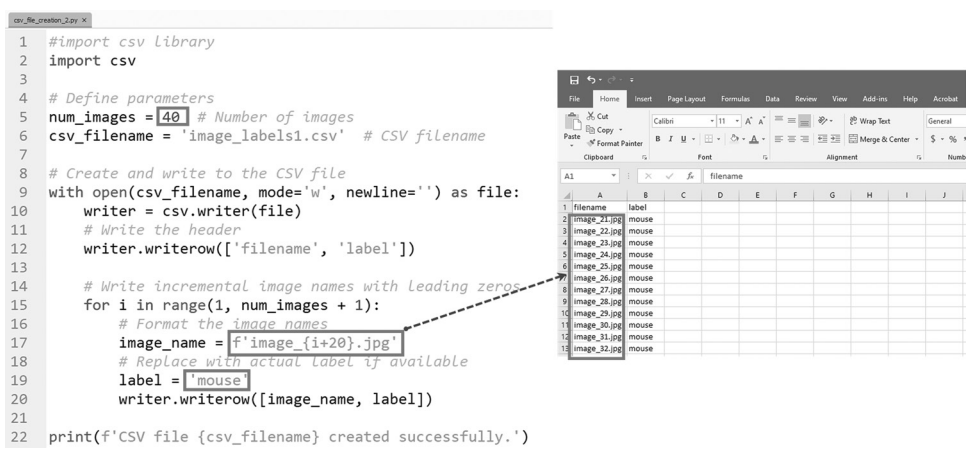


Figure 8.13 Automatic data labeling of "captured_images." Here, "image_21.jpg" to "image_60.jpg" are assigned to the label "mouse."

8.6 Data Preprocessing and Cleaning

Data preprocessing and cleaning are important steps in creating a high-quality, large dataset for deep learning. This step can significantly improve the model's performance during testing. Proper preprocessing ensures that the data is consistent, free from errors, and doesn't contain any irrelevant information. After collecting the data, it's necessary to clean it by removing any blurry, partial images or those that do not accurately represent the intended object. For image classification, organizing the image data typically involves creating a .csv file that lists all image names along with their corresponding labels or classes. If the data is intended for object detection or segmentation, further preprocessing with tools like LabelImg or Labelbox (for segmentation)

may be required. Additionally, before actual training, preprocessing techniques such as normalization, scaling, and data augmentation can improve the model's ability to learn from different scenarios.

8.7 Exercise Problem

(Q1) The "dataset1" and its corresponding annotations are included in the Chapter 9 resources. Prepare a similar customized large dataset representing five distinct classes using the labelImg and automated image capture methods discussed in this chapter. Ensure each class contains at least 100 images and label those images automatically using corresponding class names.

References

1 Chen, Y., Mancini, M., Zhu, X. et al. (2022). Semi-supervised and unsupervised deep visual learning: a survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46 (3): 1327–1347.

2 Gholizadeh, N. and Musilek, P. (2024). Daily electric vehicle charging dataset for training reinforcement learning algorithms. *Data in Brief* 55: 110587.

3 Zhu, H., Huang, M., and Zhang, Q.B. (2024). TunGPR: enhancing data-driven maintenance for tunnel linings through synthetic datasets, deep learning and BIM. *Tunnelling and Underground Space Technology* 145: 105568.

4 LabelImg (2018). *LabelImg – Graphical Image Annotation Tool*. https://github.com/HumanSignal/labelImg.

9

Training Model for Image Classification

9.1 Problem Statement

This chapter demonstrates how to train a deep learning model for image classification using a customized dataset. The dataset created in Chapter 8, named "dataset1," consists of 640 images categorized into four classes: pencil, scissors, screwdriver, and wrench. To showcase the practical application of deep learning in image classification tasks, we employed transfer learning and took advantage of using a pre-trained model.

The "dataset1" used in this chapter can be downloaded from the "chapter resources" folder available with this book. Inside the "dataset1," the "annotations" directory contains 640 XML files (named 0001.xml to 0640.xml) in Pascal VOC format that provide rectangular bounding box details for corresponding images. These annotations match the 640 images (0001.jpg to 0640.jpg) found in the "image" folder. Our goal is to develop and train a deep learning model using these labeled images and then evaluate the model's effectiveness by making inferences on unknown images. According to our dataset, a model should be able to classify any given image into one of four categories: pencil, scissors, screwdriver, or wrench. Readers are encouraged to download the dataset and follow the instructions and Python code in this chapter to replicate the results.

The pre-trained model "GoogLeNet" was utilized to classify the images through transfer learning. "GoogLeNet" is well-known for its use of the "Inception module," which allows the network to capture multi-scale features by applying multiple convolutional operations with different filter sizes. This architecture was designed to be computationally efficient and deep without using a large number of parameters [1]. It consists of twenty-two hidden layers and is trained on the ImageNet dataset. "GoogLeNet" was specifically developed to compete in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2014, where it achieved top performance in image classification tasks [2]. Figure 9.1 illustrates the transfer learning approach using the pre-trained network "GoogLeNet." After successfully training the model, we assessed its performance on various new and unseen images.

The following sections of this chapter explain the Python codes used for training and testing classification operations. The Python code in this chapter was developed and executed using the Spyder integrated development environment (IDE) from Anaconda Navigator. After launching, "Spyder" IDE may fail to detect the compute unified device architecture (CUDA) environment. We can test this by executing the "torch.cuda.is_available()" command from the "Spyder" console. This command will return "False" if CUDA is not detected. In that case, we should launch the "Spyder" IDE via the Anaconda Prompt. To do so, we have to open the Anaconda Prompt with administrator rights and enter the "spyder" command.

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

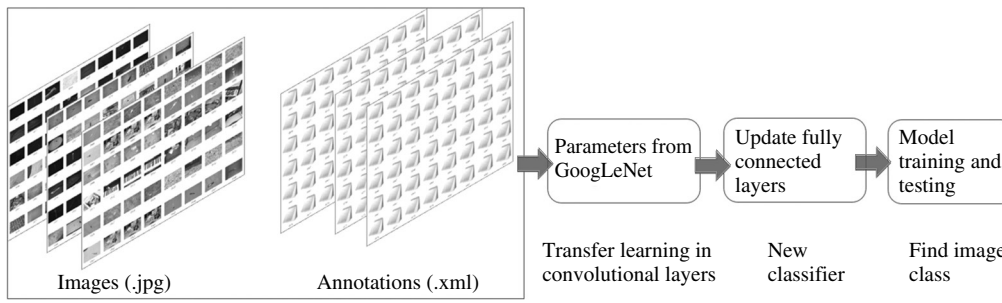


Figure 9.1 Transfer learning on a customized dataset using pre-trained network "GoogLeNet."

9.2 Default Configurations and Libraries

First of all, we define Python classes to organize parameters and paths for training. The "Class" in Python forms an object to maintain a set of attributes. Here, the "DefaultConfigs" class stores paths for input images, annotations, and the directory where the trained model will be saved. It also defines key model parameters, such as the number of classes (num_classes = 4), image dimensions (img_width = 256 and img_height = 256), and the number of channels (channels = 3). Also, the class includes hyperparameters such as the learning rate (lr = 0.001), batch size (batch_size = 16), and the number of epochs (epochs = 10). A random seed (seed = 202) is set to ensure reproducibility. By creating an instance of this class (config = DefaultConfigs()), the program allows for centralized management of these configurations during training. This approach enhances flexibility in maintenance and allows us to refer to parameter values consistently across different sections.

Next, we import necessary libraries such as "random," "numpy," and "torch." These are essential for data handling and importing deep learning frameworks. There is a deprecation for using the "np.bool" operation using the "numpy" library. To avoid errors related to this deprecation "np.bool" is set to "np.bool_." The "set_all_random_seed()" function takes a seed value (config.seed), and applies it across Python's random module, NumPy, and PyTorch (for all CUDA operations). Setting a fixed seed value (seed = 202) ensures that the training result is consistent and reproducible. In this way, the same operations performed on the "dataset1" (using the same parameters) will produce identical results. The Python codes for these operations are shown in Figure 9.2.

9.3 Setup Data Frame Using Annotations

In this step, Python's "os" for operating system interactions, "minidom" from "xml.dom" for XML document parsing, and "pandas" for data manipulation are imported. Codes for these imports are shown in the Spyder editor in Figure 9.3. A variable "annotations" is defined to store the path to the "annotations" folder. Using pandas, a data frame is created to list all files and directories within this folder. The code then iteratively processes each ".xml" file (ranging from 0001.xml to 0640.xml) from the "annotations" folder. For each file, it extracts specific XML data: the image "name" and the bounding box coordinates ("xmin," "xmax," "ymin," "ymax"). This extracted information is organized into a list called "all_data_list." For example, the XML structure of "0001.xml" have a root tag "<annotation>" with various child tags such as "<folder>," "<filename>," and "<size>." The code shown in Figure 9.3 shows how these key elements (the filename, name or label, and the four bounding box coordinates) are extracted from the XML files.

```
class DefaultConfigs(object):
    Image_PATH = r"C:\Users\tariqarif\Desktop\classification1\dataset1\images"
    Annot_PATH = r"C:\Users\tariqarif\Desktop\classification1\dataset1\annotations"
    Model_PATH = r"C:\Users\tariqarif\Desktop\classification1"
    num_classes = 4
    img_width = 256
    img_height = 256
    channels = 3
    lr = 0.001
    batch_size = 16
    epochs = 10
    seed= 202
config = DefaultConfigs()

########################
#import libraries and set random seeds
import random
import numpy as np
import torch
np.bool = np.bool_

def set_all_random_seed(seed = config.seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    return seed
print(f'current seed {set_all_random_seed()}')
```

Figure 9.2 Define the "DefaultConfigs" class to specify the parameters used across the program, import libraries, and utilize the "set_all_random_seed" function to initialize seed values.

```
#import librarie and read in data and create dataframe
import os                    # to list all files in a folder
from xml.dom import minidom  #to process the XML documents
import pandas as pd          #to create dataframe object

annotations = os.listdir(config.Annot_PATH)
all_data_list = []
for annots in annotations:
    file = minidom.parse(config.Annot_PATH+'/'+annots)
    filename = file.getElementsByTagName('filename')[0].firstChild.data
    label = file.getElementsByTagName('name')[0].firstChild.data
    xmin = file.getElementsByTagName('xmin')[0].firstChild.data
    ymin = file.getElementsByTagName('ymin')[0].firstChild.data
    xmax = file.getElementsByTagName('xmax')[0].firstChild.data
    ymax = file.getElementsByTagName('ymax')[0].firstChild.data
    all_data_list.append([filename,label,xmin,ymin,xmax,ymax])

all_data = pd.DataFrame(all_data_list)
all_data.columns = ['filename','label','xmin','ymin','xmax','ymax']
```

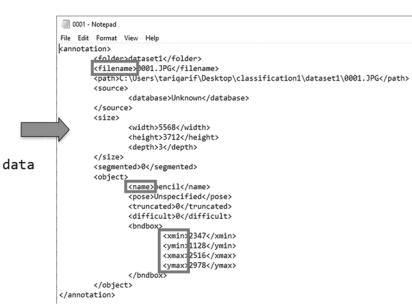


Figure 9.3 Import operating system module, minidom, and pandas, and extract information from the XML files through iterative parsing.

The extracted data are loaded into a pandas DataFrame named "all_data," with columns labeled "filename," "label," "xmin," "ymin," "xmax," and "ymax." A label encoder is then used to convert categorical data into numerical values. The "fit_transform" method of "sklearn.preprocessing" is then applied to promote convergence and reduce the risk of overshooting the minima [3].

Figure 9.4 shows how to process the labels, and prepare the dataset for training and validation. It uses "LabelEncoder" from "sklearn.preprocessing" to convert categorical labels into numerical values and stores the transformed labels in the "label1." It also creates a dictionary (le_name_mapping) to map numerical values back to their original class names. The dataset is split into training, and validation sets using the "train_test_split" function from "sklearn.model_selection." Here, with 20% of the data (randomly selected) is reserved for validation.

```
# process labels
from sklearn import preprocessing

le = preprocessing.LabelEncoder()
all_data['label1'] = le.fit_transform(all_data.label)
le_name_mapping = dict(zip(le.transform(le.classes_),le.classes_))

############################
#split the dataset into train and validation sets
from sklearn.model_selection import train_test_split

train_data_list,valid_data_list = train_test_split(all_data,test_size = 0.20,
                                                   random_state = config.seed)
############################
#prepare the dataset
from torch.utils.data import Dataset
from torchvision import transforms as T  # to transform the data into a tensor and normalize it
import cv2                               # to read image files and process
from imgaug import augmenters as iaa     # to perform image augmentation
```

Figure 9.4 Import "preprocessing" from "sklearn," apply the "fit_transform()" method, create a dictionary to validate the labels, and then randomly divide the data into training and validation sets. Additionally, import "Dataset," "transforms," "cv2," and "augmenters" libraries to preprocess data and image processing.

```
In [4]: print(all_data)
     filename   label  xmin  ymin  xmax  ymax  label1
0    0001.JPG  pencil  2347  1128  2516  2978       0
1    0002.JPG  pencil  2574   636  2816  2786       0
2    0003.JPG  pencil  1008   763  4381  2140       0
3    0004.JPG  pencil  1785  1755  3377  1982       0
4    0005.JPG  pencil  1058  1751  4150  2136       0
..        ...     ...   ...   ...   ...   ...     ...
635  0636.JPG  wrench  1797  1940  3620  2613       3
636  0637.JPG  wrench  3408  1420  4535  1701       3
637  0638.JPG  wrench  1420  1255  1908  2332       3
638  0639.JPG  wrench  3431   840  4339  1786       3
639  0640.JPG  wrench  1935  1909  3985  2390       3

[640 rows x 7 columns]
```

Figure 9.5 Verify panda data frame "all_data," and newly created labels for different classes.

The next step is to import essential libraries and modules for data management and processing. The Dataset class from "torch.utils.data" is imported to manage the dataset effectively. The "transforms" module from "torchvision" is imported to convert images into tensors and apply normalization. The "cv2" is needed to read and preprocess image files [4]. Also, "augmenters" from "imgaug" is imported to implement various image augmentation techniques such as cropping, horizontal and vertical flips, pooling, and Gaussian blur [5]. These augmentation methods help to diversify the training data and improve the model's generalization capability. If we examine the data frame "all_data," we can verify the newly assigned labels for each image. For example, "pencil" is assigned to label "0," "wrench" is assigned to label "3," and so on. Entering the "print(all_data)" command in the Spyder console will display these results, as illustrated in Figure 9.5.

9.4 Dataset Class and Methods

The custom class, "ClassificationDataset" is created where "__init__" method initializes the dataset with necessary inputs, such as DataFrame (images_df), the base path (base_path), normalization parameters (mean and std), and flags to control augmentation and training mode. The "__len__" method returns the total number of images, allowing PyTorch's "DataLoader" to iterate through the dataset.

```
class ClassificationDataset(Dataset):
    def __init__(self,images_df,base_path,mean,std,augment=True,mode="train"):
        self.images_df = images_df.copy()
        self.augment = augment
        self.mode = mode
        self.mean = mean
        self.std = std

    def __len__(self):
        return len(self.images_df)

    def __getitem__(self,index):
        X = self.read_images(index)

        if not self.mode == "test":
            y = self.images_df.iloc[index].label1

        else:
            y = (self.images_df.iloc[index].filename)

        if self.augment:
            X = self.augmentor(X)

        X = T.Compose([T.ToTensor(),T.Normalize(self.mean,self.std)])(X.copy())#pass a copy of the X here
        return X.float(),y  # convert the image to floatTensor, as model weights are in floatTensor
```

Figure 9.6 Define "ClassificationDataset" class with "init," "len," and "getitem" methods.

The "__getitem__" method retrieves an image by its index number, using the "read_images" method. If the mode is not set to "test," the corresponding label is extracted from the "DataFrame." Otherwise, only the filename is returned for testing purposes. For "augment = True," the image undergoes transformations through the "augmentor" function. After this, the image is converted to a tensor and normalized (between 0 and 1) using "ToTensor()" and "Normalize()" as shown in Figure 9.6.

The "read_images" method loads the image from the dataset using OpenCV's "imread" function. It also accesses the image file path from the "DataFrame" using "config.Image_PATH," and extracts the coordinates (xmin, ymin, xmax, ymax) to crop the image. After that, the image is normalized by dividing pixel values by 255.0 to scale them between 0 and 1. Finally, the image is resized to the specified dimensions (config.img_width and config.img_height) defined in "DefaultConfigs." The "augmentor" method improves the dataset's variability by applying data augmentation techniques. It performs a sequence of transformations, such as rotations (90, 180, and 270 degrees), shearing, and horizontal flipping to 50% images.

These transformations are applied randomly (random_order = True) to simulate different image conditions so that the model can generalize better for unseen data (Figure 9.7).

9.5 Data Loader and Model Configuration

In this classification task, we used the "GoogLeNet" architecture. To normalize the model, specific mean [0.485, 0.456, 0.406] and standard deviation [0.229, 0.224, 0.225] values are used [6]. We have utilized PyTorch's "DataLoader" from "torch" and the data loader objects have been created to iterate over the training and validation datasets. The "train_loader" is initialized with the training data generator (train_gen), using a batch size defined in "config.batch_size." The "shuffling" was enabled to ensure that the model did not rely on the order of the data during training. The "val_loader" is created for the validation dataset, but shuffling is disabled to maintain consistent evaluation. Both data loaders use "pin_memory = True" to improve data transfer speed from CPU to GPU. Detailed explanations of these loader arguments can be found in [7]. Next, we import libraries to define the pre-trained model for transfer learning. The "nn" and "optim"

```
def read_images(self,index):
    row = self.images_df.iloc[index]
    filename = str(config.Image_PATH + '/' + row.filename)
    image = cv2.imread(filename)

    xmin = int(row.xmin)
    ymin = int(row.ymin)
    xmax = int(row.xmax)
    ymax = int(row.ymax)

    cropped_image = np.array(image[ymin:ymax, xmin:xmax])
    images = cropped_image/255.0        #normalize the image

    return cv2.resize(images,(config.img_width,config.img_height))

def augmentor(self,image):
    seq = iaa.Sequential([
        iaa.OneOf([
            iaa.Affine(rotate=90),
            iaa.Affine(rotate=180),
            iaa.Affine(rotate=270),
            iaa.Affine(shear=(-16, 16)),
            iaa.Fliplr(0.5),

        ])], random_order=True)
    image_aug = seq.augment_image(image)
    return image_aug
```

Figure 9.7 The "read_images" and "augmentor" methods. Here, "read_images" crops images based on bounding box coordinates and resizes them to the desired dimensions, and the "augmentor" randomly applies one of the transformation operations.

```
# use the appropriate mean and std values that the model expects
mean=[0.485, 0.456, 0.406]
std=[0.229, 0.224, 0.225]

train_gen = ClassificationDataset(train_data_list,config.Image_PATH,mean,std,augment=True,mode="train")
val_gen = ClassificationDataset(valid_data_list,config.Image_PATH,mean,std,augment=False,mode="eval")

#########################
#create dataloader object to iterate over the datasets
from torch.utils.data import DataLoader

train_loader = DataLoader(train_gen,batch_size=config.batch_size,shuffle=True,pin_memory=True)
val_loader = DataLoader(val_gen,batch_size=config.batch_size,shuffle=False,pin_memory=True)

############################
# define the  pretrained model for tranfer learning
from torch import nn,optim
from torchvision.models import googlenet , GoogLeNet_Weights
                # may use any other models for image classifications
```

Figure 9.8 Using "DataLoader" to access training and validation datasets effectively and importing "nn," "optim" and pre-trained model "GoogLeNet."

modules, and the "GoogLeNet" architecture along with its pre-trained weights (GoogLeNet_Weights) from "torchvision.models" are also imported. If needed, the "GoogLeNet" architecture used here can be replaced by any other suitable model for image classification. The codes for these imports and configurations are shown in Figure 9.8.

Next, we need to update the pre-trained deep learning network by adjusting its input/output parameters to align with the specific classification problem. For the classification task, "torchvision" offers a variety of widely used pre-trained models, including AlexNet, VGG, ResNet variants, SqueezeNet, Inception V3, MobileNet v2, ResNeXt, Wide ResNet, and MNASNet [8]. These models can be utilized for direct inference or can be trained on customized datasets for specific tasks. The "GoogLeNet" model we plan to use for classifying objects needs to be adjusted after loading. If model loading encounters a certificate issue, it's necessary to import the "ssl" certificate to define default security settings (Figure 9.9).

The input dimension of "GoogLeNet" model is 1024 and the output dimension is 1000. This information can be verified (after loading the model) by entering "model" in the console. The last fully connected layer of the model will show "Linear (in_features=1024, out_features=1000, bias=True)." In our case, the classification problem requires four output classes (config.num_classes=4). Therefore, adjustments to the last linear layers of the model are required. To implement the adjustment, we employ PyTorch's "nn.Sequential," which creates a sequential network architecture (Figure 9.9). This new sequence has batch normalization, dropout, and a linear layer to match the number of target classes. To prevent overfitting, the architecture randomly deactivates 50% of nodes using "nn.droptout." The "nn.Linear" layer updates the input and output features of the imported model to align with our dataset's requirements. This reconstruction of the final layers prepares the pre-trained model for our specific classification task while keeping its learned features from earlier layers.

After this setup, the model is moved to the GPU for faster computation, and a loss function (CrossEntropyLoss), optimizer (SGD with momentum), and learning rate scheduler (StepLR) are defined for training. The required model parameters are set to "True" to ensure gradient calculations for every layer during the training process.

It is recommended to run each section of the program (given in this chapter) separately in the Spyder IDE. This can be done by selecting multiple lines and pressing the F9 key. Executing codes and functions in segments will be helpful for identifying and fixing errors.

```
#run if downloading model failed. model certificate issue
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

model = googlenet(weights=GoogLeNet_Weights.DEFAULT)
input_dim = model.fc.in_features
model.fc = nn.Sequential(                  #check the model and change the last
            nn.BatchNorm1d(input_dim),     #linear layer to output number of classes of the dataset
            nn.Dropout(0.5),
            nn.Linear(input_dim, config.num_classes)
        )
#model = nn.DataParallel(model) # use this for multiple GPU
model= model.cuda()

#define loss function, optimizer, and scheduler
criterion = nn.CrossEntropyLoss().cuda()
optimizer = optim.SGD(model.parameters(),lr = config.lr,momentum=0.9,weight_decay=1e-4)

scheduler = optim.lr_scheduler.StepLR(optimizer,step_size=5,gamma=0.3)

## you can manually freeze or unfreeze the weights of a model. To freeze the model set
## param.requires_grad = False
for param in model.parameters():
     param.requires_grad = True
```

Figure 9.9 Import the "GoogLeNet" model, modify fully connected layers using "nn.Sequential," and then define the loss function, optimizer, and learning rate scheduler. Also, ensure the gradient calculation by setting "requires_grad" to True.

The workstation that we used for this chapter has single graphical processing unit (GPU) (GeForce RTX 4080 SUPER). If you plan to use a workstation that has multiple GPUs, then use the command "model = nn.DataParallel." Using data parallelization will divide our training batches across all available GPUs.

9.6 Model Training

To implement the model training for image classification nested for loops are used that iterate through a specified number of epochs defined by "DefaultConfigs" (config.epochs = 10). Figure 9.10 shows the PyThon code used for this overall training process and how the best model weights are saved. The algorithm employs forward and backward propagation and calculates the loss after each iteration. Within each epoch, the model switches to training mode (model.train()), and batches of images and their corresponding labels are obtained from "train_loader." Here, inputs are moved to CUDA, and the labels are converted to a long tensor. The optimizer clears previous gradients for each batch using "optimizer.zero_grad()" and performs a forward pass to find model predictions.

```
#Train the model
steps = 0
train_loss = 0
train_accuracy = 0
best_loss= np.inf
for epoch in range(config.epochs):
    model.train()
    for images, labels in iter(train_loader):
        images, labels = images.cuda(), labels.type(torch.LongTensor).cuda() #LongTensor as the classes
                                                                              #will be integer
        optimizer.zero_grad()
        output = model.forward(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()          #keep track of the losses
        #accuracy estimation
        ps = torch.exp(output)
        equality = (labels.data == ps.max(dim=1)[1])
        train_accuracy += equality.type(torch.FloatTensor).mean()
        steps += 1
        if steps % 10 == 0:
            model.eval()
            val_loss = 0
            val_accuracy = 0
            with torch.no_grad():
                for images, labels in iter(val_loader):
                    images, labels = images.cuda(), labels.type(torch.LongTensor).cuda()
                    output = model.forward(images)
                    val_loss += criterion(output, labels).item()
                    ps = torch.exp(output)
                    equality = (labels.data == ps.max(dim=1)[1])
                    val_accuracy += equality.type(torch.FloatTensor).mean()

                if val_loss/len(val_loader) < best_loss:   #keep track of the loss, and update if decrease
                    best_loss = val_loss/len(val_loader)
                    print('saving model...')
                    torch.save(model, config.Model_PATH+'/model.pt') #if val_loss decreases, save the model
            #print the losses and accuracies
            print("Epoch: {}/{}.. ".format(epoch+1, config.epochs),
                  "Training Loss: {:.3f}.. ".format(train_loss/10),
                  "Training Accuracy: {:.3f}".format(train_accuracy/10),
                  "Validation Loss: {:.3f}.. ".format(val_loss/len(val_loader)),
                  "Validation Accuracy: {:.3f}".format(val_accuracy/len(val_loader)))
            train_loss = 0
            train_accuracy = 0
            model.train()
    scheduler.step()
print('Finish!')
```

Figure 9.10 Training loop for the classification model to obtain best weights with minimum validation loss.

```
Epoch: 8/10..  Training Loss: 0.214..  Training Accuracy: 0.919 Validation Loss: 0.077..  Validation Accuracy: 0.961
saving model...
Epoch: 8/10..  Training Loss: 0.151..  Training Accuracy: 0.962 Validation Loss: 0.068..  Validation Accuracy: 0.961
Epoch: 9/10..  Training Loss: 0.093..  Training Accuracy: 0.962 Validation Loss: 0.072..  Validation Accuracy: 0.961
Epoch: 9/10..  Training Loss: 0.084..  Training Accuracy: 0.969 Validation Loss: 0.073..  Validation Accuracy: 0.961
Epoch: 9/10..  Training Loss: 0.073..  Training Accuracy: 0.988 Validation Loss: 0.071..  Validation Accuracy: 0.961
Epoch: 10/10..  Training Loss: 0.123..  Training Accuracy: 0.956 Validation Loss: 0.076..  Validation Accuracy: 0.961
Epoch: 10/10..  Training Loss: 0.115..  Training Accuracy: 0.969 Validation Loss: 0.082..  Validation Accuracy: 0.953
Epoch: 10/10..  Training Loss: 0.061..  Training Accuracy: 0.981 Validation Loss: 0.075..  Validation Accuracy: 0.961
Epoch: 10/10..  Training Loss: 0.071..  Training Accuracy: 0.981 Validation Loss: 0.075..  Validation Accuracy: 0.969
Finish!
```

Figure 9.11 The best model achieved a training accuracy of 96.2% and a validation accuracy of 96.1%.

It then calculates the loss using the assigned criteria (criterion= nn.CrossEntropyLoss().cuda()), backpropagates the gradients (loss.backward()), and updates weight parameters (optimizer.step()).

To track the model performance, the program accumulates training losses and estimates accuracy by comparing predicted and actual labels. In the beginning, the best loss is initialized to a very large value (np.inf) so that it can be minimized to find the next best loss.

The model pauses for validation by switching to evaluation mode after every 10 steps and calculates validation loss and accuracy without updating weights. In this way, the model is saved only if the validation loss improves. After printing loss and accuracy information of an epoch they are reset and at the end of each epoch, the learning rate scheduler uses steps to adjust the learning rate.

The training process might return an "OutOfMemoryError: CUDA out of memory" error. This happens if the GPU lacks sufficient memory to handle the images. To resolve this, we can lower the batch size (e.g. from 64 to 32 or 16) or reduce the image dimensions (e.g. from 256×256 to 128×128 or 64×64). We may also test larger batch sizes and image resolutions in the "DefaultConfigs" to assess the GPU's capacity for image processing.

In this classification example, the dataset is divided into two parts: 80% for training and 20% for validation. In some machine learning practices, datasets are divided into three sets – training, validation, and testing. Although both validation and testing sets can often be used interchangeably for unbiased evaluation, the validation set is particularly useful for applying early stopping in overfitting scenarios. After training the model for 10 epochs, the minimum validation loss achieved is 0.068, resulting in a training accuracy of 96.2% and a validation accuracy of 96.1%, as shown in Figure 9.11.

It's important to monitor the model's performance during the initial epochs. If there's a trend where training accuracies increase while validation accuracies decrease, we should terminate the training process. This pattern indicates that the model is failing to generalize from the dataset and is likely overfitting.

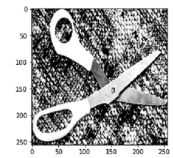
9.7 Testing and Inference

To test the model, first, the best model saved by the program is loaded from the assigned "Model_PATH" and set to evaluation mode (model.eval()). It then retrieves a batch of images and their corresponding labels from the validation dataset using the data loader. An individual image from the batch is selected based on the index number (index = 3) and displayed using "matplotlib," along with its actual label. For prediction, images are transferred to the GPU, and the model processes these inputs through a forward pass to generate output predictions. Finally, the predicted label of the image is printed and mapped to its corresponding name using the label mapping dictionary (le_name_mapping). In this program, labels 0, 1, 2, and 3 are used for "pencil," "scissors," "screwdriver," and "wrench," respectively (Figure 9.12).

```
#Load the saved model
model = torch.load(config.Model_PATH+'/model.pt')
model.eval()
##################
# Predict and display image and label.
import matplotlib.pyplot as plt

features, labels = next(iter(val_loader))
print(f"Feature batch shape: {features.size()}")
print(f"Labels batch shape: {labels.size()}")
index = 3
img = features[index].permute(1, 2, 0).numpy()
label = labels[index]
plt.imshow(img, cmap="gray")
plt.show()
print(f'Label:{label}')
#Prediction by trained model
images = features.cuda()
output = model.forward(images)
pred_labels = np.argmax(output.cpu().detach().numpy(), axis=1)

print(f"Predicted Label: {le_name_mapping[pred_labels[index]]}")
```



```
Feature batch shape: torch.Size([16, 3, 256, 256])
Labels batch shape: torch.Size([16])
```

> **Important**
> Figures are displayed in the Plots pane by default. To make them also appear inline in the console, you need to uncheck "Mute inline plotting" under the options menu of Plots.

```
Label:1
Predicted Label: scissor
```

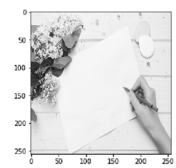
Figure 9.12 Loading the trained model, displaying the test image, and predicting its label.

```
#Testing
import matplotlib.pyplot as plt
path = r"C:\Users\tariqarif\Desktop\classification1"

image = cv2.imread(path+'/Test1.jpg')
image = image/255.0
image = cv2.resize(image,(config.img_width,config.img_height))
X = T.Compose([T.ToTensor(),T.Normalize(mean,std)])(image)
images = X.float().cuda()
images = images.unsqueeze(0)

output = model(images)
pred_labels = np.argmax(output.cpu().detach().numpy(), axis=1)

plt.imshow(image)
print(f"Predicted Label: {le_name_mapping[pred_labels[0]]}")
```



```
...: plt.imshow(image)
...: print(f"Predicted Label: {le_name_mapping[pred_labels[0]]}")
Predicted Label: pencil
```

Figure 9.13 Classify images outside of the training and validation dataset. Here, the model can correctly predict an image of a pencil. StockSnap / Pixabay.

We can also make inferences using images outside of the training and validation dataset. For this test, four open-source images were downloaded from https://pixabay.com/ and renamed as Test1.jpg, Test2.jpg, Test3.jpg, and Test4.jpg. This program shown in Figure 9.13, performs model inference by reading and preprocessing a test image to make a prediction. In this case, since we are not fetching images from "val_loader," the pixel values are normalized by dividing them by 255.0, and the image is resized to match the model's input dimensions. The image is further transformed using "ToTensor()" and "Normalize()" operations to get a tensor with standardized values. This tensor is moved to the GPU, and its shape is adjusted using "unsqueeze(0)." Finally, the model processes the input through a forward pass to generate outputs. The predicted label is determined by using the index of the highest value in the output, and the label name is printed.

Figure 9.14 shows the inference results after updating the image file names to Test2.jpg, Test3.jpg, and Test4.jpg. Although the model can make correct predictions on these uncropped images, it shows better accuracy for cropped images. The complete Python program from Figures 9.2 to 9.14 is included in the "main.py" file of chapter resources.

9.8 Fine-tuning

Optimizing parameters for deep learning is not straightforward. It is recommended to experiment with various training parameters or hyperparameters (see Chap 2.6), and assess their impact on key performance metrics such as loss function, training and validation accuracy, etc. This type of experimentation for the optimization process is known as "Hyperparameter Tuning." There's no universal set of hyperparameters for any given training task. Adjustments can involve modifying the network structure, such as altering the number of nodes, choosing different activation

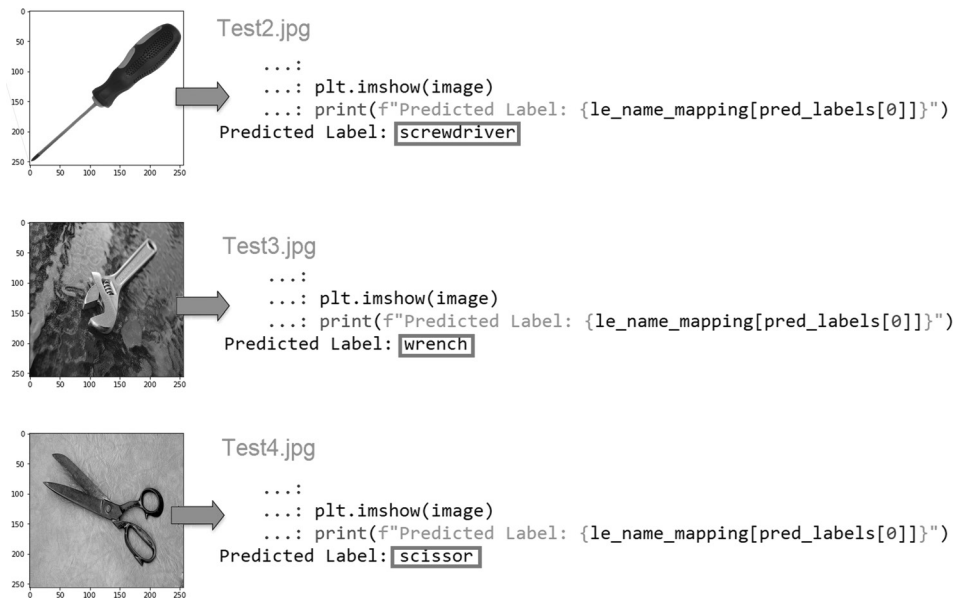


Figure 9.14 Classify images outside of the training and validation dataset. Here, the model can correctly predict images of a screwdriver, wrench, and scissors. FelixMittermeier / Public domain; ontiveros / Pixabay; Pixabay.

functions, or changing layer types. Alternatively, it may involve selecting different network components, optimizers, and loss functions. If training parameters are considered, we may use different batch sizes, learning rates, and the number of epochs. The goal is to find the ideal combination of these elements that yields the best performance for our specific deep learning task.

If we get memory-related errors during the training process, we should decrease the batch size and image shapes. Most of the time, tuning the batch size is more efficient in terms of GPU memory usage. Using a smaller batch size reduces GPU memory consumption but increases the number of iterations required. An optimum batch size can lead to more stable training compared to a larger one. A common practice is to start with a batch size of 16 for image datasets and then adjust it (e.g. 32, 64, 128) to see its impact on model performance. The number of epochs can also be adjusted to optimize training speed. Fewer epochs result in shorter training times and can enhance stability, though they may also reduce the model's prediction performance. Although more epochs generally improve performance beyond a certain point, they may no longer contribute to training and could even cause instability in the model.

The number of hidden layers and nodes per layer significantly impacts the training process and can be optimized for better results. Determining the ideal number of hidden layers is challenging, as increasing layers and nodes exponentially raises training and inference time. Also, for simple datasets, excess numbers of hidden layers and nodes may lead to overfitting.

Finding the optimal combination of hyperparameters is also dependent on intuition about data characteristics and available computational resources. When time permits, it's advisable to use grid search to systematically explore parameters within a specified range. Let's now examine the effects of hyperparameters on the training process. We selected a batch size of 16 (batch_size = 16 in DefaultConfigs class). If we use increased batch sizes (e.g. 32, 64, and 128) we observe different model performance. With a batch size of 32, the validation accuracy remains relatively stable at 96.1%, while the training accuracy drops from 96.2 to 94.7%. As the batch size increases further, we



Figure 9.15 Training and validation losses and accuracies for 32, 64, and 128 batch sizes.

observe a decline in model performance for both training and validation. Specifically, with a batch size of 128, the training accuracy falls to 73.8% and the validation accuracy to 83.6%. Additionally, when using a batch size of 128, it's necessary to reduce the image dimensions from 256 to 128 pixels to prevent the training process from becoming excessively slow. It's also worth noting that these results were obtained using a single-GPU setup, and accuracy outcomes could vary on a multi-GPU system (Figure 9.15).

Increasing the batch size also requires adjustments to the learning rate. Users should fine-tune the model by experimenting with different hyperparameters, such as batch size, learning rate, and the number of epochs. These changes are dependent on GPU capacity. In this example, with all other parameters held constant, a batch size of 16 yields better performance.

9.8.1 Using a Different Model

Experimenting with different pre-trained models in deep learning is crucial because each model brings its unique strengths and biases, which can significantly impact the performance of a given task. Different pre-trained models are trained on various datasets and architectures and have diverse feature representations. Therefore, when limited labeled data is available, we should explore various pre-trained models to find the one that best aligns with our specific problem. Additionally, some models may perform better in certain scenarios due to their architecture or the type of data they were initially trained on. In this chapter example, we have used the pre-trained model GoogLeNet. There are various models available in PyTorch that we can directly use for transfer learning. These models are available at https://pytorch.org/vision/stable/models.html. Now, let's say we would like to implement another pre-trained model Resnet50, which is a deep

```
# define the  pretrained model for tranfer learning
from torch import nn,optim
from torchvision.models import resnet50, ResNet50_Weights
                # may use any other models for image classifications

#run if downloading model failed. model certificate issue
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

model = resnet50(weights=ResNet50_Weights.DEFAULT)
input_dim = model.fc.in_features
model.fc = nn.Sequential(
            nn.BatchNorm1d(input_dim),
            nn.Dropout(0.5),
            nn.Linear(input_dim, config.num_classes)
        )
```

```
Epoch: 8/10..  Training Loss: 0.072..  Training Accuracy: 0.975 Validation Loss: 0.134..  Validation Accuracy: 0.953
Epoch: 8/10..  Training Loss: 0.082..  Training Accuracy: 0.981 Validation Loss: 0.150..  Validation Accuracy: 0.953
Epoch: 8/10..  Training Loss: 0.083..  Training Accuracy: 0.969 Validation Loss: 0.138..  Validation Accuracy: 0.953
saving model...
Epoch: 9/10..  Training Loss: 0.084..  Training Accuracy: 0.962 Validation Loss: 0.122..  Validation Accuracy: 0.953
Epoch: 9/10..  Training Loss: 0.075..  Training Accuracy: 0.969 Validation Loss: 0.138..  Validation Accuracy: 0.953
Epoch: 9/10..  Training Loss: 0.117..  Training Accuracy: 0.962 Validation Loss: 0.143..  Validation Accuracy: 0.953
Epoch: 10/10..  Training Loss: 0.059..  Training Accuracy: 0.981 Validation Loss: 0.135..  Validation Accuracy: 0.953
Epoch: 10/10..  Training Loss: 0.071..  Training Accuracy: 0.969 Validation Loss: 0.138..  Validation Accuracy: 0.953
Epoch: 10/10..  Training Loss: 0.041..  Training Accuracy: 0.988 Validation Loss: 0.125..  Validation Accuracy: 0.953
Epoch: 10/10..  Training Loss: 0.104..  Training Accuracy: 0.969 Validation Loss: 0.135..  Validation Accuracy: 0.953
Finish!
```

Figure 9.16 Using ResNet50 model with default weights.

convolutional neural network architecture and part of the ResNet (Residual Networks) family. It consists of 50 layers and uses residual connections, or "skip connections," which allow the model to learn effectively without performance degradation. In comparison to GoogLeNet, which has 22 layers (excluding pooling), ResNet50 has more layers, potentially leading to higher accuracy. This is because GoogLeNet is designed for computational efficiency, utilizing fewer parameters with the assistance of the inception module.

Figure 9.16 shows that the code section needs to be updated to implement the ResNet50 model instead of GoogLeNet. Here, we import resnet50, and its default weights from the "torchvision.models" and change the model name. We observe almost similar model performance after 10 epochs. The best validation accuracy is 95.3% compared to 96.1% on GoogLeNet, while the training accuracy remains unchanged. These results are likely to change if we run the models for more epochs (20 or 25).

9.8.2 Using Different Optimizers and Schedulers

In this example, the PyTorch's stochastic gradient descent (SGD) optimizer is employed to minimize the network's cost function. SGD is a popular optimizer for linear regression and classification tasks, and it is particularly effective with small- to medium-sized datasets. Also, PyTorch's "torch.optim" package provides a variety of other optimizers, including Adadelta, Adagrad, Adam, and RMSprop. We can get these optimizers from https://pytorch.org/docs/stable/optim.html. Every optimizer comes with its own strengths and weaknesses, and selecting the appropriate one generally depends on the complexity of the task and the size of the dataset.

To explore how the model's accuracy and loss are affected, we can experiment with different optimizers. For example, the "Adam" optimizer (uses the Adam algorithm) can be applied by importing it from PyTorch's optimization library [9]. Figure 9.17 shows the modified code and the corresponding accuracy and loss results using the "Adam" optimizer.

We notice that the "Adam" optimizer reduces losses and improves accuracy on the validation dataset. With this optimizer, the model achieves a validation accuracy of 98.4%, compared to 96.1% with the SGD optimizer.

```
#define loss function, optimizer, and scheduler
criterion = nn.CrossEntropyLoss().cuda()
#optimizer = optim.SGD(model.parameters(),lr = config.lr,momentum=0.9,weight_decay=1e-4)
optimizer = optim.Adam(model.parameters())

scheduler = optim.lr_scheduler.StepLR(optimizer,step_size=5,gamma=0.3)

## you can manually freeze or unfreeze the weights of a model. To freeze the model set
## param.requires_grad = False
for param in model.parameters():
    param.requires_grad = True
```

```
saving model...
Epoch: 6/10..  Training Loss: 0.294..  Training Accuracy: 0.913 Validation Loss: 0.088..  Validation Accuracy: 0.969
Epoch: 6/10..  Training Loss: 0.079..  Training Accuracy: 0.975 Validation Loss: 0.132..  Validation Accuracy: 0.953
Epoch: 6/10..  Training Loss: 0.120..  Training Accuracy: 0.938 Validation Loss: 0.124..  Validation Accuracy: 0.961
Epoch: 7/10..  Training Loss: 0.178..  Training Accuracy: 0.925 Validation Loss: 0.093..  Validation Accuracy: 0.969
saving model...
Epoch: 7/10..  Training Loss: 0.144..  Training Accuracy: 0.950 Validation Loss: 0.047..  Validation Accuracy: 0.984
Epoch: 7/10..  Training Loss: 0.117..  Training Accuracy: 0.969 Validation Loss: 0.050..  Validation Accuracy: 0.984
Epoch: 8/10..  Training Loss: 0.096..  Training Accuracy: 0.969 Validation Loss: 0.049..  Validation Accuracy: 0.977
saving model...
Epoch: 8/10..  Training Loss: 0.158..  Training Accuracy: 0.956 Validation Loss: 0.037..  Validation Accuracy: 0.984
Epoch: 8/10..  Training Loss: 0.093..  Training Accuracy: 0.950 Validation Loss: 0.050..  Validation Accuracy: 0.984
Epoch: 9/10..  Training Loss: 0.064..  Training Accuracy: 0.981 Validation Loss: 0.057..  Validation Accuracy: 0.977
Epoch: 9/10..  Training Loss: 0.080..  Training Accuracy: 0.969 Validation Loss: 0.047..  Validation Accuracy: 0.977
Epoch: 9/10..  Training Loss: 0.065..  Training Accuracy: 0.988 Validation Loss: 0.068..  Validation Accuracy: 0.969
Epoch: 10/10..  Training Loss: 0.069..  Training Accuracy: 0.981 Validation Loss: 0.048..  Validation Accuracy: 0.984
Epoch: 10/10..  Training Loss: 0.052..  Training Accuracy: 0.975 Validation Loss: 0.052..  Validation Accuracy: 0.984
Epoch: 10/10..  Training Loss: 0.034..  Training Accuracy: 0.994 Validation Loss: 0.046..  Validation Accuracy: 0.984
Epoch: 10/10..  Training Loss: 0.102..  Training Accuracy: 0.969 Validation Loss: 0.052..  Validation Accuracy: 0.977
Finish!

In [16]:
```

Figure 9.17 The best model using the "Adam" optimizer achieves a training accuracy of 95.6% and a validation accuracy of 98.4%.

We may consider tweaking the scheduler, along with other hyperparameters, to evaluate its impact on model performance. Initially, we used the "StepLR" scheduler with a step size of 5 and a gamma of 0.3 for learning rate decay. Other schedulers, such as MultistepLR, ExponentialLR, PolynomialLR, and CyclicLR, should be explored for better performance. It is recommended to try different schedulers and fine-tune their parameters, such as step size and gamma (for modifying the learning rate). Figure 9.18 illustrates that when the step size in the "StepLR" scheduler is increased to 8 and the gamma is changed to 0.6, the model performance does not improve. With that adjustment to the scheduler, the highest validation accuracy achieved was 94.5%.

9.9 Application in Embedded System

This chapter presents an example of image classification using deep learning. The custom image dataset utilized in this chapter is available in the "dataset1" directory of the chapter resources. Readers are strongly encouraged to run the model using this dataset. Once they become comfortable with the program structure, they should explore importing and comparing various pre-trained models for various image classification tasks.

Deep transfer learning has significant applications in embedded systems such as Jetson Nano and Raspberry Pi. Utilizing these compact, low-cost, and resource-constrained AI devices for computer vision, natural language processing, and sensor data analysis can drive innovation and new data collection approaches in the IoT field. In Chapters 12 and 15, we demonstrated how the trained classification model can be deployed on Jetson Nano and Raspberry Pi for inference. With the help of transfer learning, these devices can efficiently execute tasks such as real-time object detection, facial recognition, and image classification. These capabilities of embedded devices are valuable for enhancing navigation and decision-making in robotics, enabling predictive maintenance in factories, detecting anomalies in cybersecurity or quality control, and automating smart home systems.

```
#define loss function, optimizer, and scheduler
criterion = nn.CrossEntropyLoss().cuda()
#optimizer = optim.SGD(model.parameters(),lr = config.lr,momentum=0.9,weight_decay=1e-4)
optimizer = optim.Adam(model.parameters())

#scheduler = optim.lr_scheduler.StepLR(optimizer,step_size=5,gamma=0.3)
scheduler = optim.lr_scheduler.StepLR(optimizer,step_size=8,gamma=0.6)

## you can manually freeze or unfreeze the weights of a model. To freeze the model set
## param.requires_grad = False
for param in model.parameters():
    param.requires_grad = True
```

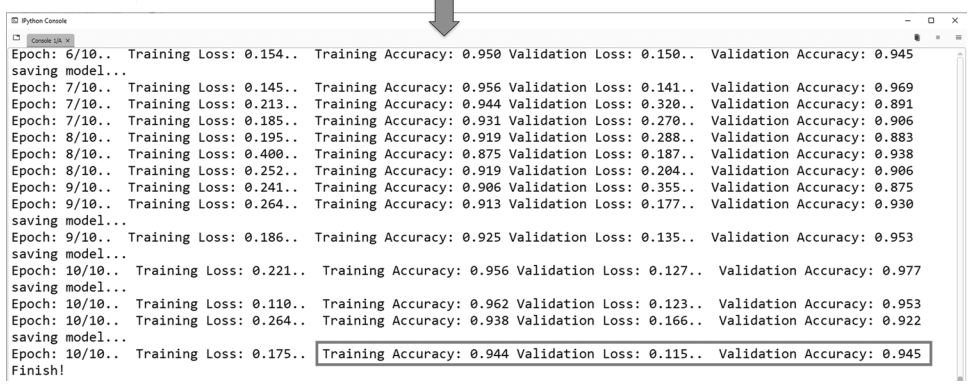


Figure 9.18 The best model using the "step_size=8" and "gamma=0.6" achieves a training accuracy of 94.4% and a validation accuracy of 94.5%.

9.10 Exercise Problems

(Q1) Utilize the dataset you created in Chapter 8 (Q1) to develop and train an image classification model through transfer learning.

(Q2) This chapter demonstrates the use of GoogLeNet and ResNet50 for training models on the "dataset1" for image classification. Use the ResNet18 model with its default weights and compare the results with those of GoogLeNet and ResNet50.

(Q3) In deep learning training, various loss functions and optimizers can be employed to evaluate model performance. The example in this chapter demonstrates the use of Cross Entropy loss (defined as "criterion") and SGD optimizers (defined as "optimizer").

Use the mean squared error (MSE) Loss in place of Cross-Entropy and the Adamax optimizer instead of SGD for training a ResNet18 model. Comment on the model performance with these adjustments.

[MSE loss is typically utilized in regression tasks, where the goal is to predict continuous values, with larger errors being penalized more heavily due to the squaring of the errors. The Adamax optimizer is a variant of the Adam algorithm (which itself is an extension of SGD), and is designed to manage cases with large gradients effectively.]

(Q4) Utilize the Multi Margin Loss function, which is specifically designed for multi-class classification tasks and is based on margin-based ranking losses. Also, modify the scheduler to "ExponentialLR" with a gamma of 0.9, which adjusts the learning rate according to an exponential decay during the training process. After applying these changes, use the pretrained GoogLeNet model for training. Evaluate the model's performance and compare it to the other scenarios discussed in this chapter.

References

1 He, K., Zhang, X., Ren, S. et al. (2015). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778. Las Vegas, NV, USA.
2 Tsang, S.H. (2018). *Review: GoogLeNet (Inception v1) – Winner of ILSVRC 2014 (Image Classification)*. https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7.
3 Pedregosa, F., Varoquaux, G., Gramfort, A. et al. (2011). Scikit-learn: machine learning in python. *Journal of Machine Learning Research* 12 (null): 2825–2830.
4 Bradski, G. (2000). The opencv library. *Dr. Dobb's Journal of Software Tools* 120: 122–125.
5 Jung, A. (2020). *Imgaug Python Library*. https://github.com/aleju/imgaug?tab=readme-ov-file#imgaug.
6 Team, P. (2014). *GoogLeNet Deep Convolutional Neural Network*. https://pytorch.org/hub/pytorch_vision_googlenet/.
7 PyTorch (2024). *Datasets & DataLoaders*. https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.
8 PyTorch (2024). *torchvision.models – Classification*. https://pytorch.org/vision/0.8/models.html.
9 PyTorch (2023). *Adam*. https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam.

10

Object Detection with Classification

10.1 Introduction

This chapter demonstrates an implementation of object detection along with image classification problems using deep learning. The Python programming and deep learning training discussed here were conducted on a desktop computer, following the setup process outlined in Chapter 4.

Object detection with image classification is relatively more challenging than regular image classification. This is because this kind of problem requires not only identifying objects within an image but also precisely locating them with bounding boxes. The objects that appear for inference can vary in size and position, and therefore, the model needs to address the variations and scale differences. Object detections with image classification problems also require us to detect overlapping or multiple objects on a single image or video frame. In this chapter, we consider a simple case similar to the problem presented in Chapter 9. We utilized the same dataset (dataset1) to classify and detect four objects: pencil, scissors, screwdriver, and wrench. The "dataset1" also has the bounding box information given in the training images as ".xml" files. In Chapter 9, we used those bounding boxes to crop the images; however, in this chapter, we will utilize them for both object detection and classification.

To address the challenges associated with detection, we can employ the Faster R-CNN with a ResNet-101 backbone, which combines the power of region proposal networks with deep convolutional features [1, 2]. The ResNet-101 can serve as the feature extractor and capture detailed hierarchical features from the input image. These features will then be used by the region proposal network (RPN) to generate bounding regions and then apply classification. The bounding box regressor can be used to detect and classify objects. Besides ResNet-101, other classification backbones can also be used for this kind of problem.

A schematic of the training process using the pre-trained network (Faster R-CNN) and ResNet-101 backbone is shown in Figure 10.1. After successful training and testing on the dataset, we used several random pictures of pencils, scissors, screwdrivers, and wrenches to evaluate the performance of the deep learning model on unseen images.

10.2 Import Modules and Libraries

First, we will need to import various libraries and modules that are required for object detection with image classification using the Faster R-CNN model. The standard Python and PyTorch libraries "os," "random," "numpy," and "torch" are used for ".csv" file handling, randomization,

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

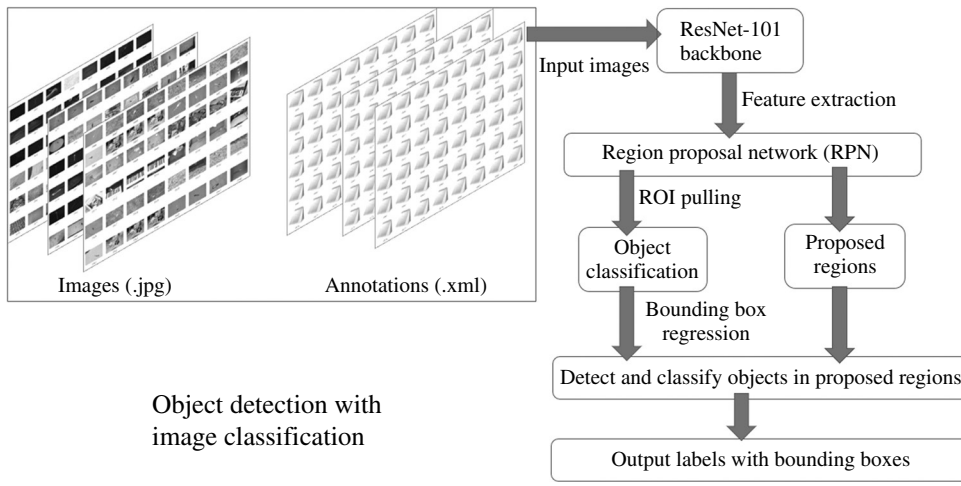


Figure 10.1 Schematic of the training process of object detection with image classification using the RPN of Faster R-CNN and ResNet-101 backbone.

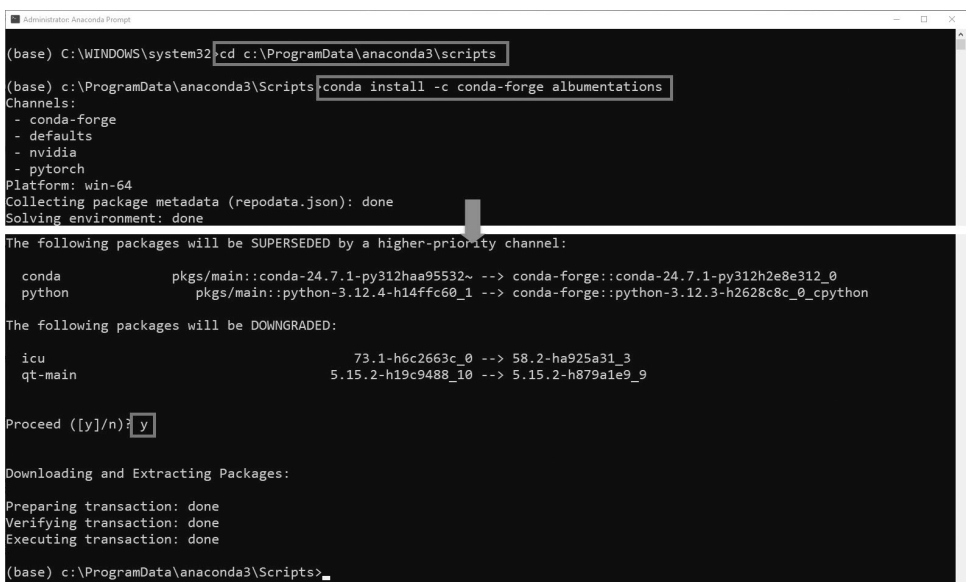


Figure 10.2 Install "albumentations" library for anaconda.

numerical computations, and deep transfer learning tasks. The "torchvision.models.detection" module that contains pre-trained models for object detections is utilized to import "FasterRCNN" and "FastRCNNPredictor." We will also need utilities for image transformations ("functional as F" from "trochvision.transforms"), dataset management utility ("Dataset" and "DataLoader"), XML parsing capability ("minidom" for annotations), and image processing (cv2). The "pandas" library is needed for data handling, and "sklearn" provides tools for data preprocessing and splitting into training and test datasets. To set up a ResNet backbone with a feature pyramid network (FPN), we used "resnet_fpn_backbone," and pre-trained weights for "ResNet-101".

main_detection.py ×

```
# import libraries
import os
import random
import numpy as np
import torch
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.transforms import functional as F
from torch.utils.data import Dataset, DataLoader
from xml.dom import minidom
import cv2
import pandas as pd
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from torchvision.models.detection.backbone_utils import resnet_fpn_backbone
from torchvision.models import ResNet101_Weights
import albumentations as A
np.bool = np.bool_
```

Figure 10.3 Import libraries and modules for object detection with classification task.

Image augmentations are managed with the "albumentations" library, and the script "np.bool = np.bool_" addresses a new deprecation warning in the current NumPy. Make sure to install the "albumentations" library if it is not available. To install this library, run "Anaconda Prompt" as an administrator and navigate to the "c:\ProgramData\anaconda3\Scripts" folder, and then enter the command "conda install -c conda-forge albumentations," as shown in Figure 10.2. After that, enter "y" to proceed.

All the library and package imports are necessary for building and training a Faster R-CNN model, and Python scripts for these imports are shown in Figure 10.3.

10.3 Default Configurations and Random Seeds

Next, we define a configuration class and set random seeds for the program. The "DefaultConfigs" class specifies essential parameters, such as file paths for images and annotations, model path, number of classes (including background), image dimensions, learning rate, batch size, and number of epochs (Figure 10.4). It also defines a seed value for reproducibility. As Faster R-CNN performs better in larger images for training, we used an image size of 800 by 800 pixels.

The "set_all_random_seed" function uses the seed number defined in the default configuration (202) to initialize random number generators for Python's random module, "NumPy," and "PyTorch" operations. This initial setup ensures the program's consistent results across various operating environments. The code also prints the current seed value that is being used.

10.4 Create Data Frame and Process Labels

To read annotation data from the ".XML" files and process it into a structured format, list all annotation files in the directory assigned at "config.Annot_PATH." The program parses the XML content from each file and extracts the "name" of the image and the bounding box coordinates ("xmin," "ymin," "xmax," "ymax") for each object, along with its label. This information is stored in a list, which is then converted into a Pandas "DataFrame" with columns for the "filename," "label," and bounding box coordinates.

```
########################
#define class objects and set random seeds
class DefaultConfigs(object):
    Image_PATH = r"C:\Users\tariqarif\Desktop\classification1\dataset1\images"
    Annot_PATH = r"C:\Users\tariqarif\Desktop\classification1\dataset1\annotations"
    Model_PATH = r"C:\Users\tariqarif\Desktop\classification1"
    num_classes = 5  # Including background
    img_width = 800 # Faster R-CNN typically uses larger image sizes
    img_height = 800
    lr = 0.005
    batch_size = 4  # Smaller batch size due to larger images
    epochs = 15
    seed = 202
config = DefaultConfigs()

def set_all_random_seed(seed=config.seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    return seed
print(f'current seed {set_all_random_seed()}')
```

Figure 10.4 Define default class objects and set random seed values.

The labels are processed using a "LabelEncoder" from "sklearn," which converts the textual labels into numeric IDs. The label IDs are incremented by 1 to reserve 0 for the background class, and a mapping between the original labels and their corresponding IDs is created. We also split the data into training and validation datasets using an 80/20 split (80% training and 20% validation). The fixed random seed ("config.seed") is used in this split to ensure reproducibility, as shown in Figure 10.5.

10.5 Training and Validation of Transformers

Data augmentation and preprocessing transformations for training and validation images are defined using the "Albumentations" library. It is a powerful and flexible tool for performing image augmentations for deep learning tasks [3]. The "get_train_transform()" function creates a composition of various

```
########################
#read data and create dataframe
annotations = os.listdir(config.Annot_PATH)
all_data_list = []
for annots in annotations:
    file = minidom.parse(os.path.join(config.Annot_PATH, annots))
    filename = file.getElementsByTagName('filename')[0].firstChild.data
    objects = file.getElementsByTagName('object')
    for obj in objects:
        label = obj.getElementsByTagName('name')[0].firstChild.data
        xmin = int(obj.getElementsByTagName('xmin')[0].firstChild.data)
        ymin = int(obj.getElementsByTagName('ymin')[0].firstChild.data)
        xmax = int(obj.getElementsByTagName('xmax')[0].firstChild.data)
        ymax = int(obj.getElementsByTagName('ymax')[0].firstChild.data)
        all_data_list.append([filename, label, xmin, ymin, xmax, ymax])

all_data = pd.DataFrame(all_data_list, columns=['filename', 'label', 'xmin', 'ymin', 'xmax', 'ymax'])
############################
# process labels
le = preprocessing.LabelEncoder()
all_data['label_id'] = le.fit_transform(all_data.label) + 1 #Add 1 because 0 is reserved for background
le_name_mapping = dict(zip(le.classes_, le.transform(le.classes_) + 1))
############################
#split the dataset into train and validation sets
train_data, valid_data = train_test_split(all_data, test_size=0.20, random_state=config.seed)
```

Figure 10.5 Read annotation data from ".XML" files, process those using "LabelEncoder()", and split the training and validation datasets.

```
# define functions for training and validation transformations
def get_train_transform():
    return A.Compose([
        A.RandomSizedBBoxSafeCrop(width=config.img_width, height=config.img_height, erosion_rate=0.2),
        A.HorizontalFlip(p=0.5),
        A.RandomBrightnessContrast(p=0.2),
        A.RandomGamma(p=0.2),
        A.GaussNoise(p=0.2),
        A.Blur(blur_limit=3, p=0.1),
        A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ], bbox_params={'format': 'pascal_voc', 'label_fields': ['labels']})

def get_valid_transform():
    return A.Compose([
        A.Resize(config.img_width, config.img_height),
        A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ], bbox_params={'format': 'pascal_voc', 'label_fields': ['labels']})
```

Figure 10.6 Functions for data augmentation and preprocessing transformations for training and validation images.

image augmentations such as random-sized cropping (while preserving bounding boxes), horizontal flipping, brightness and contrast adjustments, gamma changes, Gaussian noise, and blurring. These augmentations help increase the diversity of the training image data and can improve the model's generalization in learning. The function also normalizes the image using mean and standard deviation values of pre-trained Faster R-CNN and ResNet-101 models.

The "get_valid_transform()" function provides a simpler transformation pipeline for the validation dataset. It only resizes the images to the "img_height" and "img_weight" given in the default configuration and normalizes them. Both functions specify that bounding box annotations are in Pascal VOC format. These transformations for training and validation data are crucial for preparing the input images and increasing the model's ability to learn robust features and generalize (Figure 10.6).

10.6 Dataset Class and Methods

To initialize PyTorch's data-loading utilities and data frame, the "DetectionDataset" class with methods is defined. It initializes "DataFrame" components, such as image filenames, bounding box coordinates, and labels, along with the directory path to the images. The class provides the total number of unique images in the dataset through its "__len__" method. The "__getitem__" method retrieves a specific image and its associated annotations based on the provided index. It reads the image from the directory, converts it from BGR to RGB, and extracts the relevant bounding boxes and labels. In the "__getitem__" method, if transformations are specified, they are applied to the image and its annotations. The image is then converted to a PyTorch tensor, along with the bounding boxes and labels. Additional information, such as the image ID, area of each bounding box, and a flag indicating whether the bounding box covers multiple objects ("iscrowd"), is computed and stored in a dictionary called "target." The method finally returns the processed image tensor and the target dictionary, which can then be used by the program during training or validation. The code for the "DetectionDataset" class is given in Figure 10.7.

10.7 Data Loader and Classification Backbone

To set up the data-loading functionality for the object detection program, the "collate_fn" function is defined. It organizes batches of data in a suitable format and uses the "zip" function to group similar items (e.g. images, labels, etc.) in all samples in a batch. We then create dataset objects ("train_dataset"

```
class DetectionDataset(Dataset):
    def __init__(self, dataframe, image_dir, transforms=None):
        self.dataframe = dataframe
        self.image_dir = image_dir
        self.transforms = transforms
        self.image_names = self.dataframe['filename'].unique()

    def __len__(self):
        return len(self.image_names)

    def __getitem__(self, idx):
        img_name = self.image_names[idx]
        records = self.dataframe[self.dataframe['filename'] == img_name]

        image = cv2.imread(os.path.join(self.image_dir, img_name))
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        boxes = records[['xmin', 'ymin', 'xmax', 'ymax']].values
        labels = records['label_id'].values

        if self.transforms:
            transformed = self.transforms(image=image, bboxes=boxes, labels=labels)
            image = transformed['image']
            boxes = transformed['bboxes']
            labels = transformed['labels']

        # Convert image to PyTorch tensor if it's not already
        if not isinstance(image, torch.Tensor):
            image = F.to_tensor(image)

        # Ensure boxes and labels are tensors
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.as_tensor(labels, dtype=torch.int64)

        image_id = torch.tensor([idx])
        area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0])
        iscrowd = torch.zeros((len(records),), dtype=torch.int64)

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd

        return image, target
```

Figure 10.7 "DetectionDataset" class for effective loading, transformation, and preparation of data for object detection tasks.

and "valid_dataset") using the custom "DetectionDataset" class shown in Figure 10.7. This class takes care of data loading and preprocessing of images and their corresponding annotations. The "DataLoader" object is used to define both training and validation datasets ("train_loader" and "valid_loader"). These data loaders iterate over the datasets, apply batching and shuffling (for training data only), and use the custom "collate" function. The batch size is determined by the default configuration assigned in Figure 10.3 ("config.batch_size"). This setup enables efficient and organized loading of data during the training and validation phases (Figure 10.8).

To utilize the Faster R-CNN model with a ResNet-101 backbone for transfer learning, the "get_model" function is defined (Figure 10.9). It creates and customizes the pre-trained model to fit a specific number of object classes of our dataset. A backbone using ResNet-101 is created with a FPN, and default pre-trained weights are applied [4]. This backbone is then used to build a Faster R-CNN model. The function also updates the box predictor to match the specified number of classes by changing the number of output features.

```
def collate_fn(batch):
    return tuple(zip(*batch))

########################
#create dataloader object to iterate over the datasets
train_dataset = DetectionDataset(train_data, config.Image_PATH)
valid_dataset = DetectionDataset(valid_data, config.Image_PATH)

train_loader = DataLoader(train_dataset, batch_size=config.batch_size, shuffle=True, collate_fn=collate_fn)
valid_loader = DataLoader(valid_dataset, batch_size=config.batch_size, shuffle=False, collate_fn=collate_fn)
```

Figure 10.8 Define "collate_fn" function and create training and validation data loaders to iterate over the dataset images.

```
# function for using the  pretrained model for tranfer learning
def get_model(num_classes):
    # Create a backbone using resnet101 with FPN
    backbone = resnet_fpn_backbone(
        backbone_name='resnet101',
        weights=ResNet101_Weights.DEFAULT,  # Use the weights for ResNet-101
        )
    # Create the FasterRCNN model with this backbone
    model = FasterRCNN(backbone, num_classes=num_classes)
        # Modify the box predictor to match the number of classes
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    return model

# get class number and load the model to GPU
model = get_model(config.num_classes)
model.to('cuda')

#define optimizer and scheduler
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=config.lr, momentum=0.9, weight_decay=0.0005)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
```

Figure 10.9 "get_model" function for efficient training of a custom object detection task using the pre-trained ResNet-101 backbone, and defining optimizer and learning rate.

After defining the model, it is loaded onto the computer's graphics processing unit (GPU) for faster calculation. An optimizer is set up for using stochastic gradient descent (SGD) with learning rate, momentum, and weight decay parameters. These parameters for SGD are standard settings for training deep learning models. A learning rate scheduler (StepLR) is also defined to decrease the learning rate by a factor of 0.1 in every 5 epochs. This approach helps fine-tune and converge the model as training progresses.

10.8 Training and Validation Approach

Now, we define a function named "train_one_epoch" to train the model for one epoch. It takes several parameters as inputs, Such as the model to be trained, the optimizer for updating weights, a data loader to access batches of training data, the computation device (e.g. GPU or CPU), the current epoch number, and the total number of batches in the data loader. It is important to set the model to training mode at the beginning using "model.train()" and initialize a variable for tracking losses (total_loss) over all batches in the epoch. A for loop is used to iterate over the data loader and processing each batch of images and corresponding targets (bounding boxes, labels, etc.). The images and targets are then moved to the GPU computation device.

```
#function for training one epoch
def train_one_epoch(model, optimizer, data_loader, device, epoch, num_batches):
    model.train()
    total_loss = 0

    for i, (images, targets) in enumerate(data_loader):
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()
        # calculate losses
        total_loss += losses.item()

        if i % 10 == 0:
            print(f"Epoch {epoch+1}, Batch {i+1}/{num_batches}: Loss = {losses.item():.4f}")

    return total_loss / len(data_loader)
```

Figure 10.10 "train_one_epoch" function for training the model and finding average loss over all batches in the epoch.

For each batch, the model calculates the loss dictionary (loss_dict), which contains the losses for both the classification and bounding box regression. These losses are summed to get the total loss for a batch. The optimizer is then used to update the model's weights, and gradients are calculated with "losses.backward()", while the weights are updated with "optimizer.step()". The total loss for the epoch is updated with the loss from the current batch, and the "train_one_epoch" function prints the loss in every 10 batches to check the progress.

Finally, the function returns the average loss over all batches in the epoch. This gives an idea of the model's performance during the training process. The detailed code of this function is given in Figure 10.10.

An evaluation function, "evaluate," is defined, which takes the model, data loader, and computation device as inputs. This function is used to calculate losses from the validation dataset. First, the function sets the model to evaluation mode and then iterates through the provided data loader. Then inside "torch.no_grad()", similar to the "train_one_epoch" function, a for loop is used to iterate over the data loader, processing each batch of images and corresponding targets. The images and targets are then moved to the GPU computation device. The "evaluate" function handles two scenarios for safe operation. When the model is in training mode, it directly computes the loss, and when it is in evaluation mode, it temporarily switches to training mode to compute the loss and then switches back to evaluation mode. This approach, as shown in Figure 10.11, ensures that computation loss is consistent regardless of the model's current mode. The function accumulates the total loss over all batches and keeps a count of processed batches, returning the average loss across all evaluated samples.

10.9 Run Multiple Epochs and Save the Best

To run the model for multiple epochs while keeping track of the best-performing model based on validation loss, move the model to a GPU device. Next, initialize the number of epochs (num_epochs) for training and a variable for best validation loss (best_val_loss). Here, initially, the "best_val_loss" is set to infinity. The path to save the best model is also defined using the default model configuration path (config.Model_PATH).

```
def evaluate(model, data_loader, device):
    model.eval()  # Set the model to evaluation mode
    total_loss = 0
    count = 0

    with torch.no_grad():
        for images, targets in data_loader:
            images = list(image.to(device) for image in images)
            targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

            # In evaluation mode, the model might return predictions instead of loss
            # We need to explicitly get the loss in evaluation mode
            if model.training:
                # When in training mode, the model returns a dict with losses
                loss_dict = model(images, targets)
                losses = sum(loss for loss in loss_dict.values())
            else:
                # To compute loss during evaluation, switch to train temporarily
                model.train()
                loss_dict = model(images, targets)
                losses = sum(loss for loss in loss_dict.values())
                model.eval()

            total_loss += losses.item()
            count += 1
    # Return average loss
    return total_loss / count if count > 0 else None
```

Figure 10.11 "evaluate" function for assessing the model's performance on validation data and for monitoring training performance.

To train the model for a specified number of epochs, a for loop is set for iteration. In each epoch, the "train_one_epoch" function is called to train the model on the training data, and the "evaluate" function is used to calculate the model's performance on the validation data. After each epoch, the learning rate scheduler is stepped up to adjust the learning rate. This process helps with training convergence and fine-tuning process. The training and validation losses for the epoch are also printed to monitor progress.

The code checks whether the current validation loss is lower than the best-recorded validation loss. If it is, the current model's state is saved as the best model. This process continues until all epochs are completed. After training, the best-performing model that has minimum validation loss is saved to the directory defined by "best_model_path." Thus, the model with the best generalization to unseen validation data is retained. The Python codes for performing this training are given in Figure 10.12.

The training results are given in Figure 10.13. Although the model is run for 20 epochs, the best model with the lowest validation loss is found after 10 epochs (validation loss = 0.1025). The weights of this model were saved in the "best_model_path" directory under the name "best_faster-rcnn_model.pth."

10.10 Model Inference

For the model inference, we created a separate inference file named "main_detection_inference.py." This inference program uses some functions to process an unknown image and make predictions with bounding boxes. First, we import necessary libraries, such as "torch" for deep learning operations, "cv2" for image processing, and Matplotlib for visualization, and modules from "torchvision" for model architecture and image transformations. The "DefaultConfigs" class is defined to

```
device = torch.device('cuda')
model.to(device)

num_epochs = config.epochs
best_val_loss = float('inf')
best_model_path = os.path.join(config.Model_PATH, 'best_fasterrcnn_model.pth')

####################
#Train the model and save the best model for validation
for epoch in range(num_epochs):
    train_loss = train_one_epoch(model, optimizer, train_loader, device, epoch, len(train_loader))
    val_loss = evaluate(model, valid_loader, device)

    scheduler.step()

    print(f"Epoch {epoch+1}/{num_epochs}")
    print(f"Train loss: {train_loss:.4f}")
    print(f"Validation loss: {val_loss:.4f}")

    # Save the model only if it's the best so far
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), best_model_path)
        print(f"New best model saved with validation loss: {best_val_loss:.4f}")

print("Training complete!")
print(f"Best model saved at: {best_model_path}")
```

Figure 10.12 Run the training for multiple epochs and save the best model that has minimum validation loss.

```
Epoch 10, Batch 121/128: Loss = 0.0791
Epoch 10/20
Train loss: 0.0778
Validation loss: 0.1025
New best model saved with validation loss: 0.1025
Epoch 11, Batch 1/128: Loss = 0.0989
Epoch 11, Batch 11/128: Loss = 0.0847
Epoch 11, Batch 21/128: Loss = 0.0638
Epoch 11, Batch 31/128: Loss = 0.0805
Epoch 11, Batch 41/128: Loss = 0.0624
Epoch 11, Batch 51/128: Loss = 0.1054
Epoch 11, Batch 61/128: Loss = 0.0679
Epoch 11, Batch 71/128: Loss = 0.0868
Epoch 11, Batch 81/128: Loss = 0.0597
Epoch 11, Batch 91/128: Loss = 0.0742
Epoch 11, Batch 101/128: Loss = 0.0944
Epoch 11, Batch 111/128: Loss = 0.0779
Epoch 11, Batch 121/128: Loss = 0.0834
Epoch 11/20
Train loss: 0.0747
Validation loss: 0.1028
Epoch 12, Batch 1/128: Loss = 0.0559
```

```
Epoch 19, Batch 121/128: Loss = 0.0945
Epoch 19/20
Train loss: 0.0741
Validation loss: 0.1044
Epoch 20, Batch 1/128: Loss = 0.1029
Epoch 20, Batch 11/128: Loss = 0.1527
Epoch 20, Batch 21/128: Loss = 0.0816
Epoch 20, Batch 31/128: Loss = 0.0630
Epoch 20, Batch 41/128: Loss = 0.0676
Epoch 20, Batch 51/128: Loss = 0.0654
Epoch 20, Batch 61/128: Loss = 0.0611
Epoch 20, Batch 71/128: Loss = 0.0857
Epoch 20, Batch 81/128: Loss = 0.0826
Epoch 20, Batch 91/128: Loss = 0.0713
Epoch 20, Batch 101/128: Loss = 0.0714
Epoch 20, Batch 111/128: Loss = 0.0703
Epoch 20, Batch 121/128: Loss = 0.0685
Epoch 20/20
Train loss: 0.0746
Validation loss: 0.1056
Training complete!
Best model saved at: C:\Users\tariqarif\Desktop\classification1\best_fasterrcnn_model.pth
```

Figure 10.13 The minimum validation loss is 0.1025 after 10 epochs, and the model is saved in the "best_model_path" directory.

store configuration parameters such as the path where the model is stored and the number of classes the model was trained to detect (including the background class). The inference program also loads the pre-trained Faster R-CNN model and ResNet101_Weights for using backbones, as shown in Figure 10.14.

10.10.1 Preprocessing Functions

Next, we create functions to preprocess images for model inference by resizing them and converting them to tensors.

The "get_model" function initializes and returns a Faster R-CNN model with a ResNet-101 backbone and pre-trained weights. This function also sets the number of classes required in this inference task. The "preprocess_image" function reads an image, converts its color from BGR to RGB, and resizes it to 256 by 256 pixels. The 256 by 256 pixel image size is selected through the trial and error method. According to our test, the original size that ResNet-101 expects (224 by 224) doesn't

main_detection_inference.py

```python
#import libraries
import torch
import cv2
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from torchvision.transforms import functional as F
from torchvision.models.detection import FasterRCNN
from torchvision.models import ResNet101_Weights
from torchvision.models.detection.backbone_utils import resnet_fpn_backbone

# create class for default configuration
class DefaultConfigs(object):
    Model_PATH = r"C:\Users\tariqarif\Desktop\classification1"
    num_classes = 5  # Including background
config = DefaultConfigs()
```

Figure 10.14 Import essential libraries and modules, and define the default configuration for the model path and number of classes.

provide a good inference result for bounding boxes. The "preprocess_image" then converts the image into a PyTorch tensor with an added batch dimension for model inference. It also retains the original image size for later use.

The "rescale_boxes" function adjusts the coordinates of the predicted bounding boxes back to their original scale. This ensures that the bounding boxes align correctly with the original image dimensions. The codes for these functions are given in Figure 10.15.

```python
def get_model(num_classes):
    backbone = resnet_fpn_backbone(
        backbone_name='resnet101',
        weights=ResNet101_Weights.DEFAULT,
    )
    model = FasterRCNN(backbone, num_classes=num_classes)
    return model

def preprocess_image(image_path, target_size=(256, 256)):
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    original_size = image.shape[:2]  # (height, width)
    image_resized = cv2.resize(image, target_size)
    image_tensor = F.to_tensor(image_resized).unsqueeze(0)
    return image, image_tensor, original_size

def rescale_boxes(boxes, original_size, target_size=(256, 256)):
    y_scale = original_size[0] / target_size[0]
    x_scale = original_size[1] / target_size[1]
    scaled_boxes = boxes.copy()
    scaled_boxes[:, [0, 2]] *= x_scale  # rescale x coordinates
    scaled_boxes[:, [1, 3]] *= y_scale  # rescale y coordinates
    return scaled_boxes
```

Figure 10.15 "get_model" function to create a Faster R-CNN model with a ResNet-101 backbone, "preprocess_image" function for image processing, and "rescale_boxes" function to adjust the coordinates of the bounding boxes predicted by the model.

```
def draw_highest_score_box(image, box, class_name, score):
    plt.figure(figsize=(12, 12))
    plt.imshow(image)
    ax = plt.gca()
    x_min, y_min, x_max, y_max = box
    width, height = x_max - x_min, y_max - y_min
    rect = Rectangle((x_min, y_min), width, height, linewidth=3, edgecolor='m', facecolor='none')
    ax.add_patch(rect)
    ax.text(x_min, y_min - 10, f'{class_name}: {score:.2f}',
            color='white', fontsize=30, bbox=dict(facecolor='magenta', alpha=0.5))
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

Figure 10.16 "draw_highest_score_box" function to visually annotate an image with a bounding box around the detected object with the highest confidence score.

The "draw_highest_score_box" visually annotates an image with a bounding box around the detected object with the highest confidence score. The input to the function is an image, bounding box coordinates, class name, and the detection score. It then uses Matplotlib to display the image, adds a rectangular outline around the detected object, and places a label with the class name and confidence score near the top-left corner of the box. For clear visualization, the white label is displayed with a semi-transparent magenta background. The function turns off the image axes for a cleaner look, adjusts the layout to remove extra margins, and finally shows the annotated image (Figure 10.16).

10.10.2 Inference on Test Images

Figure 10.17 shows the object detection procedure using the trained model on a single test image (Test1.jpg) that is downloaded from https://pixabay.com. Here, within the code, we specify the class names for the objects the model can detect. If working with a dataset that includes a large number of classes (e.g. more than 10), it might be practical to create a text file in the same directory and have the inference script read the classes from that file. However, to keep this program simple, we've defined the four classes directly within the Python code. The model is then loaded from a saved state and moved to the GPU device for inference. After that, the test image goes through the "preprocess_image" function for image preprocessing. The program then runs inference on the image to obtain predictions for bounding boxes, class labels, and confidence scores. The predicted bounding boxes are rescaled using the "rescale_boxes" function to match the original image size.

In this program, we focus on the prediction with the highest confidence score only. Therefore, details of the highest confidence score, such as bounding box, class label, and scores, are used. This detection is visualized on the original image by drawing the bounding box and labeling it with the class name and confidence score. The program also prints out information about this top detection class name, confidence score, and bounding box coordinates in the console. If no detections are found, it will print a message, "No detections found," in the console. This script provides a custom pipeline for running object detection on a single image and visualizing the model output.

Figure 10.18 shows inference results on four different test images downloaded from https://pixabay.com.

The output shows that the model can predict the bounding boxes and classify the image with various confidence levels (e.g. pencil 23%, screwdriver 48%, wrench 59%, and scissors 44%).

10.11 Multiple Object Detection with Classification

The object detection method outlined in Sections 10.3–10.10 is designed to detect a single object in an image. The inference process is set up to display a box around the object with the highest classification score. However, this method won't work when multiple objects need to be classified. To demonstrate multi-object detection with classification, we will apply the same programming approach to the "dataset2" provided in this chapter's "chapter-resources" folder.

The "dataset2" consists of two directories: "images" and "annotations." The "images" directory contains 293 ".jpg" files showing random dice rolls against different backgrounds. The "annotations" directory has 293 ".XML" files that include the bounding box coordinates and labels for the dice in the corresponding ".jpg" images.

Regarding changes to the main detection program, no major modifications are necessary. However, we do need to update the "DefaultConfigs" class shown in Figure 10.4 because the number of classes has changed. Dice rolls involve six classes for the six sides, plus an additional class for the background in object detection. Therefore, we adjust the "num_classes" variable to 7. Additionally, to save the best model with the lowest validation loss, we update the model's name in the "best_model_path" variable, as illustrated in Figure 10.19.

```
# Main execution
if __name__ == "__main__":
    # Define class names
    class_names = {1: 'Pencil', 2: 'Scissors', 3: 'Screwdriver', 4: 'Wrench'}

    # Load the model
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = get_model(config.num_classes)
    model.load_state_dict(torch.load(config.Model_PATH + '\\best_fasterrcnn_model.pth',
                                     map_location=device, weights_only=True))
    model.to(device)
    model.eval()

    # Prepare the image
    image_path = config.Model_PATH + '\\Test1.jpg'  # Update this path to your test image
    original_image, input_tensor, original_size = preprocess_image(image_path)

    # Perform inference
    with torch.no_grad():
        predictions = model(input_tensor.to(device))

    # Extract predictions
    predicted_boxes = predictions[0]['boxes'].cpu().numpy()
    predicted_labels = predictions[0]['labels'].cpu().numpy()
    predicted_scores = predictions[0]['scores'].cpu().numpy()
    # Rescale boxes to original image size
    scaled_boxes = rescale_boxes(predicted_boxes, original_size)
    # Find the index of the highest score
    if len(predicted_scores) > 0:
        max_score_idx = predicted_scores.argmax()
        max_score_box = scaled_boxes[max_score_idx]
        max_score_label = predicted_labels[max_score_idx]
        max_score = predicted_scores[max_score_idx]

        # Get class name
        max_score_class_name = class_names.get(max_score_label, "Unknown")

        # Draw the box with the highest score
        draw_highest_score_box(original_image, max_score_box, max_score_class_name, max_score)

        # Print detection information
        print(f"Detected {max_score_class_name} with confidence {max_score:.2f}")
        print(f"Bounding box: {max_score_box}")
    else:
        print("No detections found.")
```

Figure 10.17 Object detection with classification inference on test images using the trained model.

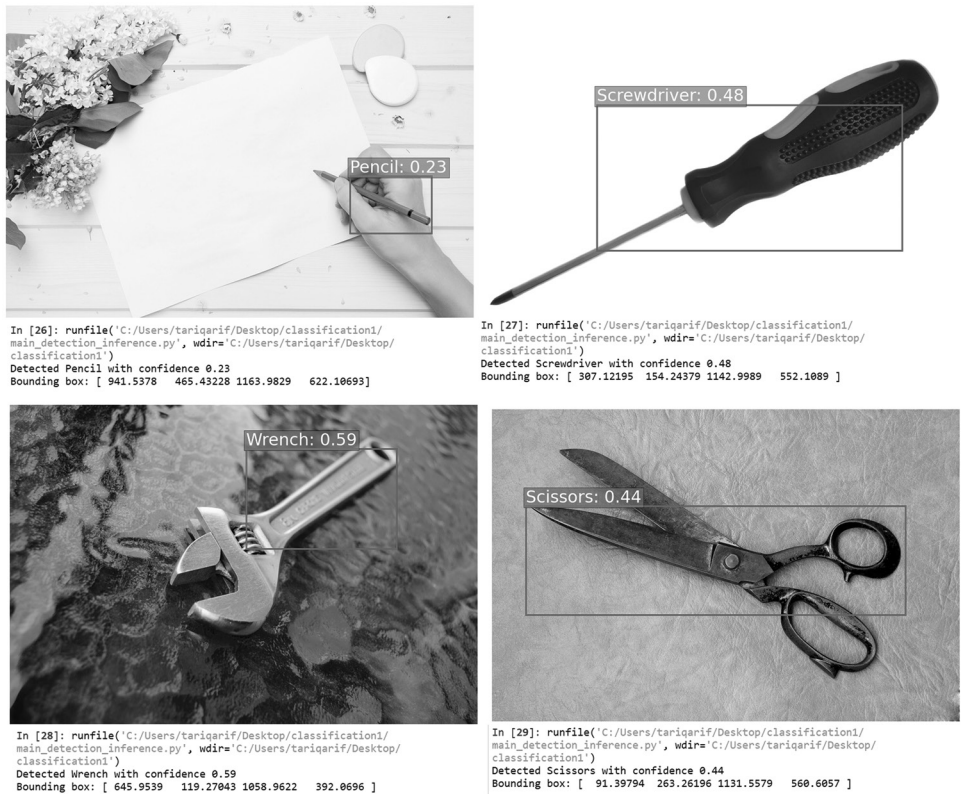


Figure 10.18 Model inference results on four different test images. StockSnap / Pixabay; FelixMittermeier / Public domain; ontiveros / Pixabay; Pixabay.

```
22  #define class objects and set random seeds
23  class DefaultConfigs(object):
24      Image_PATH = r"C:\Users\tariqarif\Desktop\classification1\dataset2\images"
25      Annot_PATH = r"C:\Users\tariqarif\Desktop\classification1\dataset2\annotations"
26      Model_PATH = r"C:\Users\tariqarif\Desktop\classification1"
27      num_classes = 7  # Including background
28      img_width = 800 # Faster R-CNN typically uses larger image sizes
29      img_height = 800
30      lr = 0.005
31      batch_size = 4  # Smaller batch size due to larger images
32      epochs = 15
33      seed = 200
34  config = DefaultConfigs()
```

```
225 device = torch.device('cuda')
226 model.to(device)
227
228 num_epochs = config.epochs
229 best_val_loss = float('inf')
230 best_model_path = os.path.join(config.Model_PATH, 'best_fasterrcnn_model2.pth')
231
```

Figure 10.19 Changes in the detection program to adjust with new class numbers in dice.

After running the program, the best model "best_fasterrcnn_model2.pth" will be saved inside the location specified by the "Model_PATH."

10.12 Model Inference for Multiple Object Detection

The model inference for multiple object detection will need to consider the existence of multiple dice in the same frame. So, we have to make some updates to the inference program. This time, we want each box to display only one class. For this, we use the non-maximum suppression (NMS) technique to refine detection results by removing redundant bounding boxes. This is a common issue in object detection tasks because, when detecting objects, the model often predicts multiple overlapping boxes for the same object, each with a different confidence score. NMS will select the bounding box with the highest confidence score and remove other overlapping boxes that have an intersection over union (IoU) above a certain threshold. This process ensures that only the most relevant bounding box for each object is retained, reducing the clutter of multiple overlapping boxes of detection results. A new function, "non_max_suppression," is defined in the inference program (main_detection2_inference.py) to perform NMS (Figure 10.20).

```
43  def non_max_suppression(boxes, scores, threshold):
44      """
45      Perform non-maximum suppression
46      find the list of indices of the boxes to keep
47      """
48      # Get coordinates of bounding boxes
49      x1 = boxes[:, 0]
50      y1 = boxes[:, 1]
51      x2 = boxes[:, 2]
52      y2 = boxes[:, 3]
53
54      # Calculate area of bounding boxes
55      areas = (x2 - x1 + 1) * (y2 - y1 + 1)
56      # Sort by confidence score
57      order = scores.argsort()[::-1]
58
59      keep = []
60      while order.size > 0:
61          i = order[0]
62          keep.append(i)
63          # Compute IoU of the picked box
64          xx1 = np.maximum(x1[i], x1[order[1:]])
65          yy1 = np.maximum(y1[i], y1[order[1:]])
66          xx2 = np.minimum(x2[i], x2[order[1:]])
67          yy2 = np.minimum(y2[i], y2[order[1:]])
68
69          w = np.maximum(0.0, xx2 - xx1 + 1)
70          h = np.maximum(0.0, yy2 - yy1 + 1)
71          inter = w * h
72
73          ovr = inter / (areas[i] + areas[order[1:]] - inter)
74          # Get indices of all boxes with IoU over threshold
75          inds = np.where(ovr <= threshold)[0]
76          # Update order
77          order = order[inds + 1]
78
79      return keep
```

Figure 10.20 "non_max_suppression" function to remove overlapping boxes for detection result.

The "non_max_suppression" function retains the most relevant bounding boxes by eliminating redundant ones. First, it extracts the coordinates (x1, y1, x2, y2) of the bounding boxes and calculates their areas. The boxes are then sorted by their confidence scores in descending order, ensuring the box with the highest score is processed first. In a while loop, it selects the box with the highest confidence score (the first box in the sorted list) and compares it with the remaining boxes by computing the IoU. The IoU measures the overlap between the selected box and other boxes. Any boxes that have an IoU above a specified threshold are removed from consideration, and the boxes with IoU below the threshold are retained. The loop continues until all relevant boxes are processed, and the function returns the indices of the boxes that should be kept, ensuring that only the most confident and non-overlapping boxes remain.

Next, we will also need to update the "draw_highest_score_box" function shown in Figure 10.16. Because this time we want multiple boxes (if applicable) on a single frame or image. We define a new "draw_boxes" function instead of "draw_highest_score_box" as shown in Figure 10.21.

The "draw_boxes" function takes the image, the coordinates of the bounding boxes, the corresponding labels, confidence scores, class names, and an optional score threshold as input. It first displays the image using Matplotlib and prepares to draw on it. For each detected object, it checks if the confidence score is above the given threshold (default is 0.5). If the score is above 0.5, it extracts the coordinates of the bounding box and draws a rectangle around the object. It then adds a label above the box, which includes the class name and the confidence score. Here, magenta color is used for both the label and the box. Finally, the function displays the image with the drawn boxes and labels.

The main execution program for inference also needs to be updated from Figures 10.17 to 10.22 to apply the NMS and draw multiple bounding boxes with classes. Figure 10.22 shows the updated code for this operation.

The inference program performs object detection on a test image (Test7.jpg) that contains dice by applying NMS, and results are visualized by drawing bounding boxes around detected objects. Here, we define class names for the objects (in this case, six possible classes representing dice faces). After the model is loaded on a GPU (if available), it is set to evaluation mode. The weights of a pre-trained model, "best_fasterrcnn_model2.pth," are loaded from "Model_PATH."

Next, we prepare the test image for inference by preprocessing it (e.g. resizing, normalization), and then the model performs inference. The model's predictions, which include bounding boxes, labels, and confidence scores, are extracted, and the code rescales them back to the original image size for proper visualization. In the code, the NMS is applied to remove redundant boxes by

```
82  def draw_boxes(image, boxes, labels, scores, class_names, score_threshold=0.5):
83      plt.figure(figsize=(12, 12))
84      plt.imshow(image)
85      ax = plt.gca()
86
87      for box, label, score in zip(boxes, labels, scores):
88          if score > score_threshold:
89              x_min, y_min, x_max, y_max = box
90              width, height = x_max - x_min, y_max - y_min
91              rect = Rectangle((x_min, y_min), width, height, linewidth=2, edgecolor='m', facecolor='none')
92              ax.add_patch(rect)
93
94              class_name = class_names.get(label, "Unknown")
95              ax.text(x_min, y_min - 10, f'{class_name}: {score:.2f}',
96                      color='white', fontsize=20, bbox=dict(facecolor='magenta', alpha=0.5))
97
98      plt.axis('off')
99      plt.tight_layout()
100     plt.show()
```

Figure 10.21 "draw_boxes" function to display inference image with bounding boxes and labels.

```
# Main execution
if __name__ == "__main__":
    # Define class names
    class_names = {1: 'One', 2: 'Two', 3: 'Three', 4: 'Four', 5: 'Five', 6: 'Six'}
    # Load the model
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = get_model(config.num_classes)
    model.load_state_dict(torch.load(config.Model_PATH + '\\best_fasterrcnn_model2.pth',
                                     map_location=device, weights_only=True))
    model.to(device)
    model.eval()
    # Prepare the image
    image_path = config.Model_PATH + '\\Test7.jpg'  # Update this path to your test image
    original_image, input_tensor, original_size = preprocess_image(image_path)
    # Perform inference
    with torch.no_grad():
        predictions = model(input_tensor.to(device))
    # Extract predictions
    predicted_boxes = predictions[0]['boxes'].cpu().numpy()
    predicted_labels = predictions[0]['labels'].cpu().numpy()
    predicted_scores = predictions[0]['scores'].cpu().numpy()
    # Rescale boxes to original image size
    scaled_boxes = rescale_boxes(predicted_boxes, original_size)
    # Perform nonmaximum suppression
    nms_threshold = 0.3  # Adjust this value as needed
    keep_indices = non_max_suppression(scaled_boxes, predicted_scores, nms_threshold)
    # Filter predictions based on suppressions
    filtered_boxes = scaled_boxes[keep_indices]
    filtered_labels = predicted_labels[keep_indices]
    filtered_scores = predicted_scores[keep_indices]
    # Draw boxes for all predictions above a certain threshold
    score_threshold = 0.3  # You can adjust this threshold
    draw_boxes(original_image, filtered_boxes, filtered_labels, filtered_scores, class_names, score_threshold)
    # Print detection information
    for box, label, score in zip(filtered_boxes, filtered_labels, filtered_scores):
        if score > score_threshold:
            class_name = class_names.get(label, "Unknown")
            print(f"Detected {class_name} with confidence {score:.2f}")
            print(f"Bounding box: {box}")
    if len(filtered_scores) == 0 or all(score <= score_threshold for score in filtered_scores):
        print("No detections found above the threshold.")
```

Figure 10.22 Multiple object detection with classification using NMS and the trained model.

keeping only the boxes with the highest confidence scores for each object based on an IoU threshold. The remaining bounding boxes, labels, and scores are filtered according to the NMS results. The "draw_boxes" function is then utilized to visualize the filtered predictions on the original image. Finally, the program prints out information about the detections, including the class name, confidence score, and bounding box coordinates. If no detections are above the defined score threshold, it prints a message indicating that no detections were found. Figure 10.23 shows the inference result with the "Test7.jpg" image. Here, we see that the model can predict the dice faces fairly well using multiple bounding boxes.

Figure 10.24 shows the inference results on the "Test8.jpg," "Test9.jpg," and "Test10.jpg" images. For these inferences, we used a threshold score of 0.3. However, this parameter needs to be fine-tuned if the model encounters issues detecting objects accurately due to image quality or background noise.

10.13 Conclusion

Training a model for object detection combined with classification is quite challenging. The process is highly sensitive to hyperparameters such as batch size, image size, optimizer, and scheduler. The programs presented in this chapter carefully tune these hyperparameters through trial and error to achieve the best possible results. The detection and inference programming part given in Sections 10.11 and 10.12 would also work for a situation when we have only one item (similar to dataset1) in the picture.

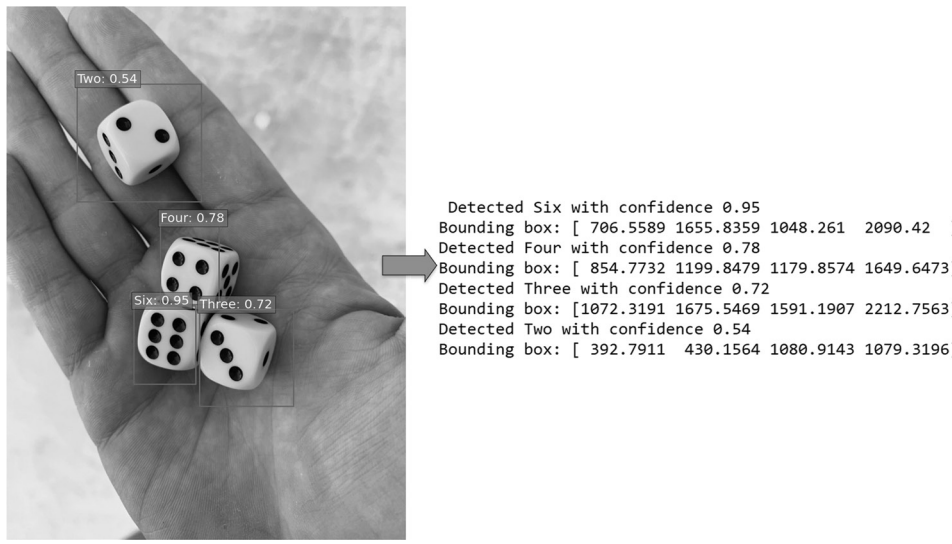


Figure 10.23 Dice face detections using the trained "best_fasterrcnn_model2.pth" model.

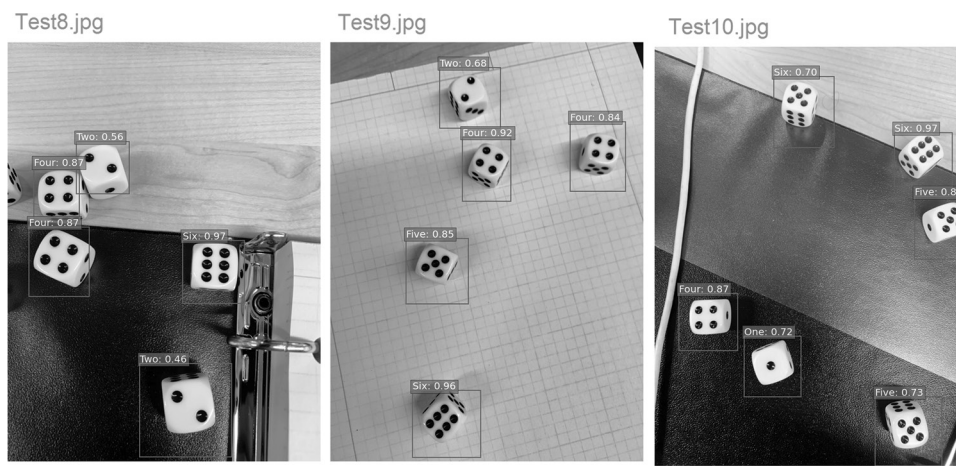


Figure 10.24 Dice face detections of test images using the trained "best_fasterrcnn_model2.pth" model.

While the bounding box component generally performs well, the classification part struggles to converge when using smaller image sizes. Additionally, the learning rate chosen in the program has a significant impact on model performance. It is recommended to experiment with different learning rates to assess their effects on performance. Also, exploring various input parameters for the optimizer (e.g. learning rate, momentum, weight decay) and scheduler (e.g. step size, gamma) is crucial for fine-tuning the model.

10.14 Exercise Problems

(Q1) The Python program presented in this chapter uses the faster RCNN model with the ResNet-101 backbone. The ResNet-101 backbone can also be utilized for its few trainable layers. In this exercise, modify the program to use a ResNet-50 backbone with 640 by 480 image sizes instead and compare the results when utilizing it with only three trainable layers versus without any trainable layers. Does this lead to an improvement in model performance?

(Q2) Use the ResNet-152 model as a backbone without trainable layers and compare the performance with ResNet-101 and 52. How does the image size affect model performance?

(Q3) In question 2, how does reducing the image size slightly (e.g. 224×224) and increasing the batch size impact the model's performance?

(Q4) Create a dataset similar to the "dataset2" (review the dataset development from Chapter 8 if needed) that has multiple objects in the same image. Train an object detection program for this using a faster RCNN model with a ResNet-101 backbone. After training, perform inference using a sample image.

For reference, review the Python code given in "main_detection2.py" and "main_detection2_inference.py".

References

1 He, K., Zhang, X., Ren, S. et al. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

2 Ren, S., He, K., Girshick, R. et al. (2015). Faster R-CNN: towards real-time object detection with region proposal networks. *Proceedings of the 28th International Conference on Neural Information Processing Systems*, *Volume 1*: 91–99. Montreal, Canada: MIT Press.

3 Buslaev, A., Iglovikov, V.I., Khvedchenya, E. et al. (2020). Albumentations: fast and flexible image augmentations. *Information* 11 (2): 125.

4 Lin, T.Y., Dollár, P., Girshick, R. et al. (2017). Feature pyramid networks for object detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 936–944.

11

Deploy Deep Learning Models on Jetson Nano

11.1 Introduction

Low-power embedded systems became useful for deep learning tasks in the mid-2010s. This is due to the advancements in compact hardware, complex neural network applications, and the increasing demand for edge devices. Deploying models on embedded systems like the Jetson Nano is important as it enables real-time inference in various portable areas and can improve security by keeping the data processing task local. Typically, embedded systems have significantly lower power consumption compared to cloud-based solutions. They are particularly valuable in applications such as autonomous vehicles, robotics, smart home devices, and the Internet of Things (IoT). There are pre-trained models (ResNet, MobileNet, GoogLeNet, etc.) that can be utilized on embedded systems using the frameworks such as PyTorch and TensorFlow. These models come with optimized weights for various tasks, such as convolutional neural networks (CNNs) for image processing, recurrent neural networks (RNNs) for sequence data, and transformer models for natural language processing [1]. In PyTorch, pre-trained models are accessible through libraries such as "torchvision," "torchaudio," and "Hugging Face" Transformers library. Embedded systems also enable developers to take advantage of cutting-edge pre-trained models despite having limited computational power. In this chapter, we'll demonstrate how to download and use a pre-trained model from the "torchvision" library for inference and how to convert a PyTorch "pt" model to an "ONNX" model for faster inference speed on embedded systems.

11.2 Pre-trained Models

In the embedded system, it is convenient to save a pre-trained model in the local storage rather than loading the model every time we run the inference. The "torchvision" library has a variety of pre-trained models that have been trained on large datasets and optimized for tasks such as image classifications, object detection, semantic segmentation, video classification, etc. These models are useful for transfer learning, where we can fine-tune the pre-trained models to achieve high performance with less training time. In this chapter, we will download and use a pre-trained ResNet18 model for inference without making any modifications.

To save a pre-trained deep learning model using PyTorch, first import the "torch" and "torchvision.models" libraries. The codes shown in Figure 11.1 are written using the VS code editor in the Jetson Nano, and the Python file named "model_download.py" is saved in the "home/jetson/Desktop/deploy-models" directory. After importing the necessary libraries, load the ResNet-18 model that has been

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

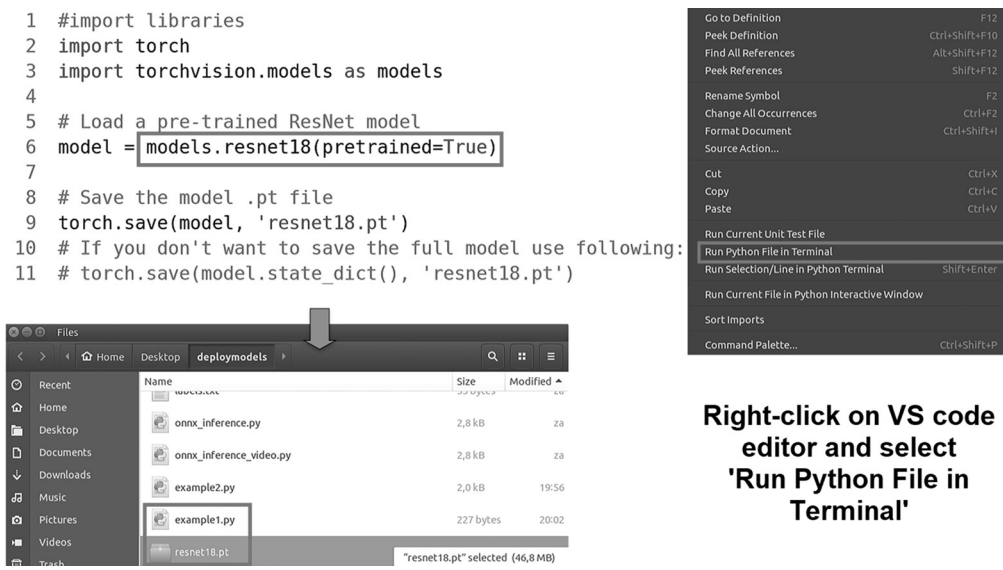


Figure 11.1 Load and save the "Resnet-18" pre-trained model (resent) in the current directory.

pre-trained on the ImageNet dataset and save the model in the same directory using "torch.save(model, "resnet18.pt")." If we only need to save the model's state dictionary, which contains the learned parameters (weights and biases) without the model's architecture, use "model.state_dict()" in place of "model" in the "torch.save" function. Right-click on the VS code editor and select "Run Python File in Terminal" to execute the code. Make sure that the Jetson Nano is connected to the internet during execution. An Ethernet cable is preferred over a Wi-Fi connection to have a smooth and fast download.

11.3 Inference on an Image File

The "example1.py" script (from Figures 11.2–11.4) is used to evaluate the ".pt" model using a test image file. To perform this inference, we'll need to import the "cv2," "torch," "torchvision.models," "Image" from the "PIL" and "numpy" libraries. Load the pre-trained model from the same directory and set it to evaluation mode. Here, the model is loaded using "torch.load," which is recommended for embedded systems. Alternatively, if internet connectivity and microSD storage are available, the model can also be directly loaded from the "torchvision" library using "resnet18 = models.resnet18(pretrained=True)".

Next, set the preprocessing steps for the input image. "transforms. Resize(256)" resizes an image to 256 pixels on its shorter side and maintains the aspect ratio. "transforms.CenterCrop(224)" crops the center portion of the resized image to get an input image pixel dimension that "resnet-18" expects (224×224 pixels). "transforms.ToTensor()" converts the image to a PyTorch tensor and scales pixel values from 0 to 255 range. The "transforms.Normalize()" normalizes the tensor image by subtracting the mean and dividing by the standard deviation for each color channel. These specific mean and standard deviation values are used for normalization because they were calculated from the ImageNet dataset, on which ResNet-18 was trained. This step converts the input image to a distribution similar to the training data.

For the model inference, we load a car image from the same directory (deploymodels). This is an open-source image downloaded from https://pixabay.com. The image is then preprocessed, which

example1.py

```
# Import necessary libraries
import cv2
import torch
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image
import numpy as np

# Load a pre-trained ResNet18 model
resnet18 = torch.load('/home/jetson/Desktop/deploymodels/resnet18.pt')
# Alternatively, you can use: resnet18 = models.resnet18(pretrained=True)
resnet18.eval()  # Set the model to evaluation mode

# Define image preprocessing steps
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

Figure 11.2 Import libraries, load the pre-trained model, and preprocess the input image for inference.

resizes, crops, converts to a tensor, and normalizes the image. Since PyTorch models typically expect input in batches, the image tensor is expanded with an additional batch dimension using "torch.unsqueeze(img_t, 0)." The model then performs inference on the image within a "torch.no_grad()," which disables gradient calculation for efficiency since the model is already trained. The output "out" is the model's prediction, containing scores for each possible class, and the class with the highest score is determined using "torch.max," and its index is stored in "predicted_index."

Next, the code reads the ImageNet class names from a local text file, "imagenet_classes.txt," into a list. The "predicted" class index is used to retrieve the corresponding class name from this list, and finally, the predicted class index and name are printed to show the classification result (Figure 11.3).

```
# Load and preprocess the input image
img = Image.open("/home/jetson/Desktop/deploymodels/car.jpg")
img_t = preprocess(img)
batch_t = torch.unsqueeze(img_t, 0)  # Add batch dimension

# Perform inference
with torch.no_grad():
    out = resnet18(batch_t)
    _, indices = torch.max(out, 1)
    predicted_index = indices.item()

# Read the ImageNet class names from the local file
with open("/home/jetson/Desktop/deploymodels/imagenet_classes.txt", "r") as f:
    classes = [line.strip() for line in f.readlines()]

# Get the predicted class name
predicted_class = classes[predicted_index]
print(f'Predicted class index: {predicted_index}, Class name: {predicted_class}')
```

Figure 11.3 Load and preprocess a test image, perform inference, read "imagenet" class names, and get the classification using a predicted index value.

```
# Convert the PIL image to an OpenCV image (numpy array)
img_cv2 = np.array(img)
img_cv2 = cv2.cvtColor(img_cv2, cv2.COLOR_RGB2BGR)
# Define the font and text properties
font = cv2.FONT_HERSHEY_SIMPLEX
text = f'{predicted_class}'
font_scale = 2  # font scale
thickness = 3   # thickness
text_color = (255, 0, 255)  # Magenta color in BGR
# Calculate text size and position
text_size = cv2.getTextSize(text, font, font_scale, thickness)[0]
text_x = (img_cv2.shape[1] - text_size[0]) // 2
text_y = img_cv2.shape[0] - 40  # Move text up a bit

# Put the text on the image
cv2.putText(img_cv2, text, (text_x, text_y), font, font_scale, text_color, \
    thickness, cv2.LINE_AA)
# Display the image with the predicted class label
cv2.imshow('Predicted Image', img_cv2)

# Wait until a key is pressed, then close the image window
cv2.waitKey(0)
cv2.destroyAllWindows()
# Save the image with the class name overlay
cv2.imwrite("/home/jetson/Desktop/deploymodels/car_with_class_cv2.jpg", img_cv2)
```



Figure 11.4 Convert the image into OpenCV-compatible format, define text properties, and overlay it on the output image. Noel_Bauza / Pixabay.

To display the output classification over the test image, use Python's OpenCV library. First, convert the PIL format image to an OpenCV-compatible image through the "numpy" library. We also defined the font type, text for predicted class, font scale, thickness, and color for the overlay text. Also, using cv2, the code determines text size and horizontal central position, which is slightly above the bottom edge of the image. The text is written on the image using "cv2.putText," and the image with the overlay text is displayed in a window using "cv2.imshow." After executing the program, it will wait for a key press to close or destroy the output window. The image with the predicted class label will also be saved to the same directory on the Jetson Nano. These steps are shown in Figure 11.4.

11.4 ONNX Model

Open neural network exchange (ONNX) is an open-source format that represents machine learning models. It defines a common file format that enables AI developers to use models with a variety of frameworks, tools, runtimes, and compilers [2]. ONNX is a community project that was initially developed by Facebook (now Meta) and Microsoft in 2017 to create a standardized way of representing AI models [3]. The interoperability of ONNX is especially valuable for developers using PyTorch, as it assists them in training models in PyTorch and then exporting them to ONNX format for deployment in embedded systems. Additionally, ONNX supports optimized inference engines, such as ONNX Runtime or TensorRT, which can efficiently run these models by applying techniques like graph optimization, layer fusion, and precision reduction to speed up inference. In this section, we will provide a Python script to convert a trained PyTorch model (in .pt format) to an ONNX model, which will then be used for inference on a video stream from a USB camera.

11.4.1 Convert PyTorch Model to ONNX

To convert the downloaded ResNet-18 model to ONNX format, first import the torch and "torch.onnx" library and set the model's file path to "/home/jetson/Desktop/deploymodels" in the Jetson Nano directory. "torch.load" loads the "resnet18.pt" model from the model path, and "model.to(device)" sets the model to evaluation mode. The code also checks if a GPU is available, and if so, it assigns the model to the GPU; otherwise, it uses the CPU. A dummy input tensor simulating an input image with a shape of (1, 3, 224, 224) and a batch size of 1, three color channels (RGB), and 224×224 pixel dimensions is created and moved to the device.

pt2onnx.py

```
#import libraries
import torch
import torchvision.models as models
import torch.onnx

# Set the path
Model_PATH = r"/home/jetson/Desktop/deploymodels"

# Load the pre-trained ResNet18 model
model = torch.load(Model_PATH+'/resnet18.pt')
model.eval()

# Check if a GPU is available and move the model to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Define dummy input# ResNet18 expects 224x224 size images
dummy_input = torch.randn(1, 3, 224, 224).to(device)

# Export the model to ONNX format
torch.onnx.export(model,
                  dummy_input,
                  Model_PATH+'/resnet18.onnx',
                  export_params=True,
                  opset_version=11,
                  do_constant_folding=True,
                  input_names=['input'],
                  output_names=['output'])

print(f"Model has been converted to ONNX and saved at {Model_PATH}")
```

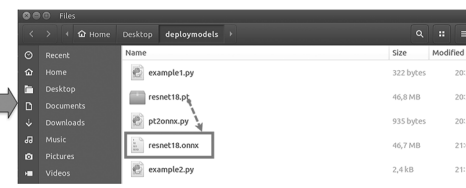


Figure 11.5 Import libraries, load pre-trained "resnet18.pt" model, define dummy input, and prepare the mode for exporting to ONNX format.

The "torch.onnx.export" function converts the model specified by the model variables. The function takes several parameters, such as "dummy_input," "Model_PATH," "export_params," "opset_version," "do_constant_folding," and input/output names. The "dummy_input" represents a sample input tensor to trace the model's operations, and Model_PATH+ "/resnet18.onnx" specifies the file path where the ONNX model will be saved. The "export_params=True" argument ensures that model parameters are included in the export. "opset_version=11" sets the ONNX operator set version, and "do_constant_folding=True" enables optimization operations. "input_names" and "output_names" define labels for the model's input and output tensors that need to be called during inference. After the export, a print statement confirms the successful conversion and saving of the model, as shown in Figure 11.5. This process converts the PyTorch's ".pt" to an ONNX that can be deployed on different ONNX-compatible hardware.

11.5 Inference on Live Video Stream

Figure 11.5 also shows that a model named "resnet18.onnx" is saved in the "deploymodels" directory in the Jetson Nano. Since the ONNX model is relatively faster than PyTorch's native format, in this section, we deploy the "resnet18.onnx" for inferencing on a video stream.

An ONNX model can be run through ONNX runtime, optimizing, and accelerating the inference of ONNX models. However, installing ONNX runtime on a Jetson Nano can be challenging as it requires specific versions of libraries like OpenCV and Numpy. It may cause conflicts with the existing installations of PyTorch, OpenCV, and NumPy. As an alternative, we utilized the Jetson inference library, which can load and run ONNX models through NVIDIA's high-performance TensorRT library. TensorRT is designed to optimize models for Jetson's specific GPU architecture, which results in faster and more efficient inference. This optimization process includes graph

optimizations, precision adjustments (e.g. FP16 or INT8), and other performance enhancements suitable for Jetson's hardware.

Figure 11.6 shows the Python codes for importing OpenCV, NumPy, Jetson inference, and Jetson Utils libraries. The "cv2" is for working with the camera and image processing, "numpy" is used for numerical operations, and "jetson_inference" and "jetson_utils" are used for running inference. The ONNX model for image classification is loaded using the "jetson_inference.imageNet" class, which also specifies the model file path, the labels file with class names, and the names of the input and output blobs in the network. The program opens a connection to a USB camera using OpenCV's "VideoCapture" function and captures video frames from the camera. This code prepares the system to perform real-time image classification on video frames.

Next, we create a continuous while loop for capturing frames from the USB camera and resize the frames to 224 × 224 pixels, as expected by the ResNet18 model. The OpenCV's "cvtColor" function converts the captured image format from BGR to RGBA. The frame is then converted to a CUDA image for efficient processing on the GPU. To perform inference and classify each frame, the loaded ONNX model is used. It obtains the predicted class index and confidence score using the "GetClassDesc()" function (Figure 11.7). Using a loaded text file, "imagenet_class.txt," and the class index, the class name is retrieved, and the confidence percentage is calculated. The output confidence scores from the Jetson inference range from 0 to 20. So, we normalized this range by multiplying it with (100/20) to display the confidence level from 0 to 100%. To display the classification result text (class name and confidence) on the video frame, OpenCV's text rendering functions are utilized.

The text box size is calculated using the "cv2.getTextSize()" function to ensure the overlay text fits correctly. The text is positioned in the top-left corner of the frame, and a filled black rectangle is applied behind it to enhance visibility. The text is then rendered on the frame using the "cv2.putText()" function. Finally, the "cv2.imshow()" displays the processed output frame, which contains the overlay text titled "Resnet18 Inference."

The infinite loop continues to do inference operations until the "q" key is pressed, at which point the loop breaks. After that, to release the video camera and close all OpenCV windows, "captured_video.

example2.py ×

```
#import libraries
import cv2
import numpy as np
import jetson_inference
import jetson_utils

# Load the ONNX model with image net class files
net = jetson_inference.imageNet(
    model="/home/jetson/Desktop/deploymodels/resnet18.onnx",
    labels="/home/jetson/Desktop/deploymodels/imagenet_classes.txt",
    input_blob="input",
    output_blob="output"
)

# Open the USB camera (adjust the index if necessary)
captured_video = cv2.VideoCapture(0)
```

Figure 11.6 Import libraries, load ONNX model, labels, and input/ output names for the inference, and capture videos using cv2's "VideoCapture" function.

```
# Continuous while loop
while True:
    # Capture a frame from the camera
    ret, frame = captured_video.read()
    if not ret:
        print("Failed to capture frames")
        break
    # input size, resnet18 expect 224 by 224 image size
    frame_resized = cv2.resize(frame, (224, 224))
    # Convert the frame to RGBA format for Jetson Inference
    img_rgba = cv2.cvtColor(frame_resized, cv2.COLOR_BGR2RGBA)
    # Convert the frame to CUDA image
    img_cuda = jetson_utils.cudaFromNumpy(img_rgba)
    # Perform the inference
    class_idx, confidence = net.Classify(img_cuda)
    # Get the class name directly from the net object
    class_name = net.GetClassDesc(class_idx)
    # format the confidence as a percentage.
    # Output confidence range from jetson inference is 0 to 20
    confidence_percentage = (confidence/20)*100
    # Create the overlay text
    overlay_text = f"{class_name} ({confidence_percentage:.2f}%)"
    # Define the font, size, and color
    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 2
    font_color = (255, 0, 255)  # Magenta color for the text
    font_thickness = 2
```

Figure 11.7 Use a continuous while loop to capture frames, apply classification, and display results.

release()" and "cv2.destroyAllWindows()" are used. These programming steps are shown in Figures 11.7 and 11.8. Figure 11.8 also shows the inference results on a video stream where the ONNX model can correctly predict a mouse and ballpoint pen with 73.55 and 67.55% accuracy, respectively.

11.6 Conclusion

The ResNet-18 is a highly optimized model, and we can implement it on embedded devices effectively by converting it to ONNX format. While our tests indicate that the ONNX model is very fast, there are some losses in detection accuracy when performing inference using the Jetson inference library. The process of converting a trained PyTorch model to ONNX and conducting inference on an embedded system shown in this chapter is also applicable to customized models that are trained on different datasets. Chapter 12 will demonstrate this process using a customized model (trained on a desktop PC) which was originally developed in Chapter 9. The setup described in this chapter enables real-time visual feedback of classification results on a captured video stream using a fast ONNX model. We encourage readers to try the Python code provided in this chapter on a Jetson Nano device, which can be configured according to the approach shown in Chapter 5.

```
    # Get the size of the text box
    (text_width, text_height), baseline = cv2.getTextSize(overlay_text, \
        font, font_scale, font_thickness)
    # Define the text position in the top-left corner
    text_x = 20
    text_y = text_height + 20
    # Draw a background rectangle for the text
    cv2.rectangle(frame, (text_x, text_y - text_height - 5),
                  (text_x + text_width, text_y + baseline),
                  (0, 0, 0), thickness=cv2.FILLED)
    # Overlay the text on the image
    cv2.putText(frame, overlay_text, (text_x, text_y), font, font_scale, \
        font_color, font_thickness, lineType=cv2.LINE_AA)
    # Display the frame with the overlay text
    cv2.imshow("Resnet18 Inference", frame)
    # Press 'q' key to exit the loop
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
# Release the camera and close all windows
captured_video.release()
cv2.destroyAllWindows()
```

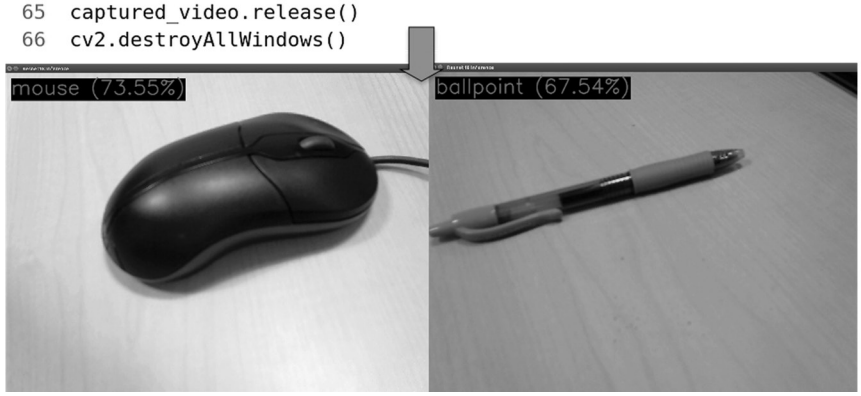


Figure 11.8 ONNX model inference using the Jetson inference library.

11.7 Exercise Problem

(Q1) Download a pre-trained classification model onto your Jetson device and convert it to the ONNX format. Then, perform inference on a video frame by following the instructions provided in this chapter. Be sure to also download the classification label (.txt) file for the model.

References

1 PyTorch (2024). *Models and Pre-Trained Weights*. https://pytorch.org/vision/stable/models.html.
2 ONNX (2019). *Open Neural Network Exchange*. https://onnx.ai.
3 Xu, F. (2018). *Onnx Runtime for Inferencing Machine Learning Models Now in Preview*. https://azure.microsoft.com/en-us/blog/onnx-runtime-for-inferencing-machine-learning-models-now-in-preview.

12

Trained PyTorch Model: From Desktop PC to Jetson Nano

12.1 Introduction

There are instances where we need to transfer a trained PyTorch model from a desktop PC to embedded devices such as Jetson Nano and Raspberry Pi. This is particularly useful when carrying powerful laptops or computers for real-time inference is impractical, such as in moving vehicles, drones, or mobile machinery. Since embedded devices often have limited computational power, converting a PyTorch model (stored as a .pt or .pth file) to the open neural network exchange (ONNX) format simplifies this process. The ONNX format is more portable than the ".pt" format and provides an optimized model representation that can be easily deployed across various edge computing devices. Additionally, it can be further optimized using TensorRT for faster inference. This chapter will demonstrate a simple example of this model transfer. We will first create an ONNX model from the custom classification model introduced in Chapter 9, and then use this ONNX model for inference on the Jetson Nano.

12.2 Model Training on a PC

The model training process using deep transfer learning is demonstrated in Chapter 9, where we used a dataset containing four different classes (pencil, scissors, wrench, and screwdriver) for an image classification problem. The custom model created in that chapter is saved inside the "classification1" directory and named "model.pt." For reference on this training process, review Chapter 9.

12.3 ONNX Model Inference

12.3.1 PyTorch's.pt to ONNX Conversion

To convert the PyTorch model to ONNX format, first, import the "torch" and "torch.onnx" modules. The location where the model is stored is specified using "Model_PATH." In this case, the model is saved in the "/home/jetson/Desktop/pc2jetson" directory on the Jetson Nano. However, this model conversion can also be performed on the desktop PC where the model was originally trained, which would likely be faster. The model is then loaded from the "pc2jetson" directory using "torch. load." The code checks if the model is wrapped in "DataParallel," which is necessary if multiple

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

graphical processing units (GPUs) were used for training on the desktop PC. In this way, it will be able to extract the underlying model with "model.module," if needed. Then the model is switched to evaluation mode using "model.eval()," which turns off certain layers, such as dropout, to ensure consistent inference results. The code also checks for GPU availability and moves the model to the GPU for faster processing. If no GPU is found, the model remains on the CPU.

After loading the model onto the GPU, the code proceeds to export the "model.pt" file. It utilizes a dummy input tensor with dimensions (1, 3, 256, 256), which simulates a single RGB image with a size of 256 × 256 pixels. This specific dimension is used because the model was originally trained on images of that size. The "torch.onnx.export" function is then used to convert the PyTorch model to ONNX format. This function requires several inputs: the actual ".pt" model, the dummy input, the output file path, export parameters, the opset version, constant folding for optimization, and specified names for input and output tensors. Running this process on a Jetson device may take 2–4 minutes. Once the conversion is complete, the code prints a confirmation message indicating where the ONNX model is saved. This conversion allows the model to be compatible with ONNX runtimes and ensures more efficient deployment on embedded devices. The codes for this conversion are given in Figure 12.1.

example1.py

```
#import libraries
import torch
import torchvision.models as models
import torch.onnx

# Set the path
Model_PATH = r"/home/jetson/Desktop/pc2jetson"

# Load the pre-trained classification model
model = torch.load(Model_PATH+'/model.pt')
#Check if the model is wrapped in DataParallel
if isinstance(model, torch.nn.DataParallel):
    model = model.module
model.eval()

# Check if a GPU is available and move the model to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Define dummy input# The model expects 256x256 size images
dummy_input = torch.randn(1, 3, 256, 256).to(device)

# Export the model to ONNX format
torch.onnx.export(model,
                  dummy_input,
                  Model_PATH+'/model.onnx',
                  export_params=True,
                  opset_version=11,
                  do_constant_folding=True,
                  input_names=['input'],
                  output_names=['output'])

print(f"Model has been converted to ONNX and saved at {Model PATH}")
```

Figure 12.1 Python code to convert a trained PyTorch model to ONNX format.

12.3.2 Image Classification Using ONNX

After having the ONNX (model.onnx) model, our next step is to utilize this model in Jetson Nano for real-time inference. If the ONNX model conversion is done on a desktop pc, we can copy that model to Jetson Nano using a flash drive. In our case, we stored the ONNX model inside "/home/jetson/Desktop/pc2jetson" directory of Jetson Nano. In the same directory, we created another Python file using the visual studio (VS) code editor named "example2.py."

First, we import "cv2" for video capture, "jetson_inference," and "jetson_utils" libraries for using Jetson's inference capabilities. We will need to use the "jetson.inference" and "jetson.utils" modules instead of "jetson_inference" and "jetson_utils", respectively, if we encounter an import error. We then load the converted ONNX model along with its corresponding labels file, using the "jetson_inference.imageNet" class. The "input_blob" and "output_blob" parameters specify the names of the input and output layers in the model. This is necessary for inferencing the data correctly. The program captures video from a USB camera connected to the Jetson Nano, using OpenCV's "VideoCapture" function, as shown in Figure 12.2. We have used the camera's default index (VideoCapture(0)), but this value needs to be adjusted if multiple cameras are connected to Jetson.

Next, we created a continuous while loop to capture frames from the video source, preprocess each frame, and perform inference using a pre-trained classification model. After capturing the video frame, it is resized to 256 × 256 pixels to match the model's input requirements. The frame is then converted from BGR to RGBA color format to work with the Jetson inference library and transferred to the GPU using compute unified device architecture (CUDA). The "net.Classify()" method performs the actual inference on the CUDA image and returns the predicted class index and confidence score. We then retrieve the human-readable class name using "net.GetClassDesc()" and format the confidence as a percentage by multiplying it by 10. We also prepared an overlay text with the classification result and confidence, which can be displayed on the video frame (Figure 12.3).

The text box size is calculated using the "cv2.getTextSize" function. The font type, scale, and thickness are also defined. The text is then positioned in the top-left corner of the frame. A white rectangle is drawn behind the text using "cv2.rectangle" to ensure visibility, regardless of the existing background. The text is then overlaid onto the frame with "cv2.putText." The processed frame after inference has the overlaid text that shows the image class is shown continuously using "cv2.imshow." The while loop checks for the "q" key to be pressed, and if true, it breaks and ends the inference video, as shown in Figure 12.4.

example2.py

```
1  # import libraries
2  import cv2
3  import numpy as np
4  import jetson_inference
5  import jetson_utils
6
7  # Load the ONNX model with the labels file
8  net = jetson_inference.imageNet(model="/home/jetson/Desktop/pc2jetson/model.onnx",
9                                  labels="/home/jetson/Desktop/pc2jetson/labels.txt",
10                                 input_blob="input",
11                                 output_blob="output")
12 # capture video from USB camera (adjust index : default = 0)
13 captured_video = cv2.VideoCapture(0)
```

Figure 12.2 Import libraries, load ONNX model, and capture video to perform real-time inference on the video feed from the USB camera.

```
# continuously process video frames
while True:
    # Capture a frame from the camera
    ret, frame = captured_video.read()
    if not ret:
        print("Failed to capture image.")
        break
    # Resize the frame to the model's expected input size
    frame_resized = cv2.resize(frame, (256, 256))
    # Convert the frame BGR to RGBA format for Jetson Inference
    img_rgba = cv2.cvtColor(frame_resized, cv2.COLOR_BGR2RGBA)
    # Convert the frame to CUDA image
    img_cuda = jetson_utils.cudaFromNumpy(img_rgba)
    # Perform the inference
    class_idx, confidence = net.Classify(img_cuda)
    # Get the class name directly from the net object
    class_name = net.GetClassDesc(class_idx)
    # Correctly format the confidence as a percentage
    confidence_percentage = confidence * 10

    # Create the overlay text to display result
    overlay_text = f"{class_name} ({confidence_percentage:.2f}%)"

    # Define the font, size, and color with reduced text size
    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 2  # Adjust the font size as needed
    font_color = (255, 0, 255)  # Magenta color for the text
    font_thickness = 4  # Define font thickness
```

Figure 12.3 A continuous while loop for processing frames in real-time and performing model inference.

We also release the camera resources using "release()" and close OpenCV windows using "destroyAllWindows()."

```
    # Get the size of the text box
    (text_width, text_height), baseline = cv2.getTextSize(overlay_text, font,\
        font_scale, font_thickness)

    # Define the text position in the top-left corner
    text_x = 15  # 15 pixels from the left
    text_y = text_height + 15  # 15 pixels from the top
    # Draw a white background rectangle behind the text
    cv2.rectangle(frame, (text_x, text_y - text_height - 5),
                  (text_x + text_width, text_y + baseline),
                  (255, 255, 255), thickness=cv2.FILLED)

    # Overlay the text on the image
    cv2.putText(frame, overlay_text, (text_x, text_y), font, font_scale, \
        font_color, font_thickness, lineType=cv2.LINE_AA)

    # Display the resultant frame
    cv2.imshow("Inference Video Using ONNX Model", frame)
    # Check for the 'q' key to exit the loop
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the camera and close all windows
captured_video.release()
cv2.destroyAllWindows()
```

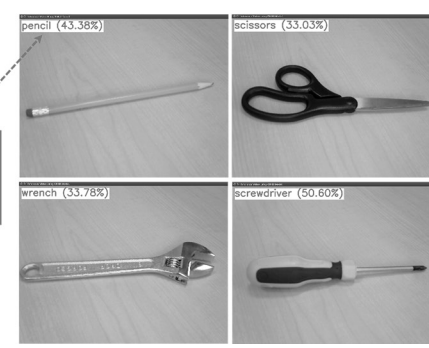


Figure 12.4 Real-time inference using the ONNX model. Here, the output results are displayed on the video stream using OpenCV, and the inference process can be exited by pressing "q."

12.3.3 PyTorch's.pth to ONNX Conversion

The object detection with classification models we created in Chapter 10 is in ".pth" format. These models are the same as ".pt" models that can save states. ".pth" model extensions are typically used to save parameters without the underlying model architecture. In this section, the ".pth" model (best_fasterrcnn_model.pth) that was created for detecting and classifying pencils, scissors, wrenches, and screwdriver objects will be used. To convert the PyTorch model to ONNX format, we used a Python program from a desktop PC named "pth2onnx.py." The configurations on the desktop PC are the same as illustrated in Chapter 4.

After importing the necessary libraries from "torch" and "torchvision," we will need to load the pre-trained model "best_fasterrcnn_model.pth" from the "Desktop/pc2jetson" directory. The model's parameters are loaded in the CPU using the "torch.load" function. Next, Faster R-CNN model is created with a ResNet-101 backbone. This matches the architecture used during training. Based on the trained model's architecture, we also need to set the number of classes to five (four object classes and one background class). After this, the model is switched to evaluation mode using "model.eval()" as shown in Figure 12.5.

To convert the PyTorch ".pth" model to "ONNX" format, we created a dummy input tensor with dimensions matching the expected input size of the trained model (1, 3, 800, 800). Then, "torch.onnx.export()" is utilized to perform the conversion by specifying the model name, dummy input, output file path, ONNX operator set version, input/output tensor names, and dynamic axes for batch size. The "torch.onnx.export()" function also sets the names for the output tensors (e.g. boxes, labels, and scores) and allows dynamic batch sizes for both input and output (Figure 12.6). After executing the program in Windows, the model will be created and saved. Conversion to the ONNX format enhances the model's portability and interoperability with various operating systems and deployment platforms.

12.3.4 Object Detection Using ONNX

To check the performance of the object detection ONNX file, we transfer the "best_fasterrcnn_model.onnx" to the Jetson Nano. It is saved inside the "pc2jetson" folder. For this step, we will need to make sure that "onnxruntime" is installed on the device. We may run the following command in Jetson Nano's terminal:

```
$ sudo apt-get update
$ pip3 install onnxruntime
```

```
#import necessary libraries
import torch
from torchvision.models.detection.faster_rcnn import FasterRCNN
from torchvision.models.detection.backbone_utils import resnet_fpn_backbone
#define Model_path
Model_PATH = r'C:\\Users\\tariqarif\\Desktop\\pc2jetson'
#Load state dictionary to cpu
state_dict = torch.load(Model_PATH+'/best_fasterrcnn_model.pth', map_location=torch.device('cpu'))

# FasterRCNN model with ResNet101 backbone
backbone = resnet_fpn_backbone('resnet101', pretrained=False)
model = FasterRCNN(backbone, num_classes=5)  # Change this num_classes based on the trained model
                                             # We have 4 classes + 1 for background
# Load the state dictionary into the model
model.load_state_dict(state_dict)
# Set the model to evaluation mode
model.eval()
```

Figure 12.5 Import libraries, load the state dictionary into the model, and set the model to evaluation mode.

```
# Create a dummy input similar to the input image size of the trained model
#The object detection model we trained expects 800 by 800 pixels
dummy_input = torch.randn(1, 3, 800, 800)

# Export the model from .pth to ONNX
torch.onnx.export(model, dummy_input, Model_PATH+"/best_fasterrcnn_model.onnx",
                  opset_version=11, # ONNX opset version
                  input_names=['input'],
                  output_names=['boxes', 'labels', 'scores'],
                  dynamic_axes={'input': {0: 'batch_size'},#allow dynamic dimensions for batch size
                                'boxes': {0: 'batch_size'},
                                'labels': {0: 'batch_size'},
                                'scores': {0: 'batch_size'}})

print(f"ONNX model is created and saved at {Model_PATH}")
```

Figure 12.6 Create dummy input similar to the input of the trained model and export the model from ".pth" to ".onnx" format.

In Jetson's "pc2jetson" folder, we created a Python file named "example3.py" to run inference using the ONNX model. This time, we'll be performing inference on test images, as loading large (800 by 800) input frames for video inference would be computationally expensive for the Jetson Nano. First, we import necessary libraries such as "onnxruntime" for inference, OpenCV for image processing, and "numpy" for numerical operations. We also suppress warnings in the output and set the logging level low. Then we define "Model_PATH" and "Image_PATH" to access the ONNX model and the image. The confidence threshold is defined to set the minimum confidence level to 0.2 for detections. The non-maximum suppression (NMS) threshold is set to 0.2 to filter overlapping detections. The input image needs to be resized to 800 × 800 pixels before being passed to the model, as the model is trained on 800 × 800 images. The maximum display size is also set to 800 for proper visualization.

Next, we map class labels to specific RGB colors that will be used for drawing bounding boxes around detected objects. These steps are shown in Figure 12.7.

The image processing function "preprocess_image" is created for image processing and visualization. It reads an image, resizes it to a specified input size (800 by 800), converts it to RGB, normalizes pixel values, and transposes the dimensions. These steps are required to match the test image size with the model's expectation. The "non_max_suppression" function is very critical, and it applies the NMS operation to remove overlapping bounding boxes. The "draw_boxes" function is used to visualize the detected objects in the image. It scales the bounding boxes to match the original image size, applies the confidence threshold to filter out low-confidence results, implements NMS, and produces detections. This function also utilizes a scale factor to ensure consistent drawing of bounding boxes for various image sizes as shown in Figure 12.8.

For each detection, the "draw_boxes" function extracts the bounding box coordinates, labels, and scores, then rescales the coordinates to match the original image dimensions. The function then retrieves class names and corresponding colors for detected objects. The thickness of the bounding boxes and text font sizes on the output images are scaled dynamically to ensure consistent appearance across different image sizes. Using OpenCV, the function then draws bounding boxes and adds labels with the class name and confidence score. It finally appends each detection (class name and score) to a list. This process can visualize all valid detections on the input image while tracking what was detected by the model (Figure 12.9).

The "resize_for_display" resizes an image while maintaining its aspect ratio if it exceeds the defined maximum size (800 by 800). Then we initialize an ONNX Runtime inference session using the model path and CPU execution providers. The input image is preprocessed using the "preprocess_image" function. This function returns the image, an input tensor, and the original image size. Finally, the inference is performed by running the ONNX model with the prepared input tensor.

example3.py

```python
# Import necessary libraries
import onnxruntime as ort
import cv2
import numpy as np
import warnings
# Suppress warnings and set ONNX Runtime logger severity
warnings.filterwarnings("ignore")
ort.set_default_logger_severity(3)

# Define paths and constants
Model_PATH = "/home/jetson/Desktop/pc2jetson/best_fasterrcnn_model.onnx"
Image_PATH = "/home/jetson/Desktop/pc2jetson/Test1.jpg"
Confidence_THRESHOLD = 0.2
# Confidence thresold. Finetune it if needed for the test environment
#images from various dataset or sources will require finetuining.
#increasing the confidence threshold may help eliminate overlapping detections.
NMS_THRESHOLD = 0.2  # Non-Maximum Suppression threshold. Finetune this value if needed
Input_SIZE = (800, 800)  # Input size for the model
MAX_DISPLAY_SIZE = 800  # Maximum size for display
Class_NAMES = {1: 'Pencil', 2: 'Scissors', 3: 'Screwdriver', 4: 'Wrench'}

# Define colors using RGB values
COLORS = {
    'Pencil': (255, 0, 0),      # Red
    'Scissors': (0, 255, 0),    # Green
    'Screwdriver': (0, 0, 255), # Blue
    'Wrench': (255, 255, 0),    # Yellow
    'Unknown': (128, 128, 128)  # Gray
}
```

Figure 12.7 Import libraries, suppress warnings, define paths, thresholds, and color mapping based on detected objects.

```python
# Define image preprocessing function
def preprocess_image(image_path, input_size):
    image = cv2.imread(image_path)
    original_size = image.shape[:2]
    image_resized = cv2.resize(image, input_size)
    image_rgb = cv2.cvtColor(image_resized, cv2.COLOR_BGR2RGB)
    image_normalized = image_rgb.astype(np.float32) / 255.0
    image_transposed = np.transpose(image_normalized, (2, 0, 1))
    return image, image_transposed[np.newaxis, ...], original_size
# Non Max Suppression to remove overlapping boxes
def non_max_suppression(boxes, scores, labels, threshold):
    indices = cv2.dnn.NMSBoxes(boxes.tolist(), scores.tolist(), Confidence_THRESHOLD, threshold)
    return indices
# Draw bounding boxes and detect
def draw_boxes(image, boxes, labels, scores, original_size):
    h, w = original_size
    h_ratio, w_ratio = h / Input_SIZE[1], w / Input_SIZE[0]
    # Calculate scaling factor for consistent drawing
    scale_factor = min(h, w) / 800  # Assuming 800 for the base size
    # Use confidence threshold
    mask = scores > Confidence_THRESHOLD
    boxes = boxes[mask]
    labels = labels[mask]
    scores = scores[mask]
    # Apply Non Maximum Suppression
    indices = non_max_suppression(boxes, scores, labels, NMS_THRESHOLD)

    detections = []
```

Figure 12.8 Image, Non-Max Suppression, and bounding box processing functions for object detection.

```
    for i in indices:
        i = i[0] if isinstance(i, (list, np.ndarray)) else i
        box = boxes[i]
        label = labels[i]
        score = scores[i]
        x1, y1, x2, y2 = box
        x1, y1, x2, y2 = int(x1 * w_ratio), int(y1 * h_ratio), int(x2 * w_ratio), int(y2 * h_ratio)
        class_name = Class_NAMES.get(int(label), "Unknown")
        color = COLORS.get(class_name, COLORS['Unknown'])

        # thickness and font size adjustment
        thickness = max(1, int(3 * scale_factor))
        font_scale = max(0.5, 1 * scale_factor)

        # Draw main bounding box
        cv2.rectangle(image, (x1, y1), (x2, y2), color, thickness)
        label_text = f'{class_name}: {score:.2f}'

        # Use updated font size and thickness for the output
        cv2.putText(image, label_text, (x1, y1 - 5), cv2.FONT_HERSHEY_SIMPLEX, font_scale, color, thickness)
        detections.append((class_name, score))

    return image, detections
```

Figure 12.9 "draw_boxes" function adjusts bounding box thickness and font size to scale output images dynamically.

After inference, the program extracts the output tensors containing bounding boxes, class labels, and confidence scores for the detected objects. These output shape tensors are printed in the console. The program also shows the number of initial detections and the top 5 confidence scores without NMS. These intermediate outputs would help understand the raw output of the object detection model before post-processing (Figure 12.10).

```
# Resize the image but keep aspect ratio
def resize_for_display(image, max_size):
    h, w = image.shape[:2]
    if max(h, w) > max_size:
        if h > w:
            new_h, new_w = max_size, int(max_size * w / h)
        else:
            new_h, new_w = int(max_size * h / w), max_size
        return cv2.resize(image, (new_w, new_h))
    return image
# Initialize ONNX Runtime
session = ort.InferenceSession(Model_PATH, providers=['CPUExecutionProvider'])

# Preprocess the input image
original_image, input_tensor, original_size = preprocess_image(Image_PATH, Input_SIZE)
# Inference
input_name = session.get_inputs()[0].name
outputs = session.run(None, {input_name: input_tensor})

# Print output shapes (optional) and number of detections
print("Outputs shapes:", [out.shape for out in outputs])
boxes = outputs[0]
labels = outputs[1]
scores = outputs[2]

print("Boxes shape:", boxes.shape)
print("Labels shape:", labels.shape)
print("Scores shape:", scores.shape)
print(f"Number of initial detections: {len(boxes)}")
print(f"Best 5 scores before NMS: {scores[:5]}")
```

Figure 12.10 Initialize ONNX runtime and run inference for object detection.

The "draw_boxes" function is utilized to visualize detected objects on the original image using bounding boxes. After applying NMS and confidence thresholding, the final detections are determined at this stage. It returns the annotated image and a list of final detections. Our output prints these final detections and shows the class names with confidence scores for each object that passed the filtering. Next, it resizes the resulting image for display using the "resize_for_display" as shown in Figure 12.11. Using OpenCV's "imshow" function, the final annotated image is shown in a window titled "Object Detection with classification by ONNX." The program waits for a key press before closing the display window, and a full-resolution annotated image is saved to a file named "detection_with_classification.jpg" in the same directory.

Figure 12.12 displays the inference results on three test images (Test1.jpg, Test2.jpg, and Test3.jpg). In the object detection program, it is very critical to manipulate the confidence score and Non-Max Suppressions (currently, both are 0.2) to get a better result. Typically, the model performance varies significantly if images are collected from various sources, using different camera resolutions and different lighting conditions. For a professional dataset, it is very important to use the same image-capturing environment to create a robust dataset. In Figure 12.12, we see that different colors are used to detect different objects as defined in Figure 12.7.

Figure 12.13 shows two more inference results on test images with different sizes. In this case, even though the image sizes are different, the program is able to process them and display the outputs (rescaled bounding boxes and labels) on the original images.

```
# Draw bounding boxes and get final detections
result_image, final_detections = draw_boxes(original_image, boxes, labels, scores, original_size)
# Final detections by NMS
print("\nFinal detections after NMS:")
for i, (class_name, score) in enumerate(final_detections, 1):
    print(f"{i}. {class_name}: {score:.2f}")

# Display and save the result
display_image = resize_for_display(result_image, MAX_DISPLAY_SIZE)
cv2.imshow('Object Detection with classification by ONNX', display_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
cv2.imwrite('detection with classification.jpg', result image)
```

Figure 12.11 Draw bounding boxes and get detections by Non-Max suppressions.



Figure 12.12 Object detection using ONNX model (best_fasterrcnn_model.onnx) in Jetson Nano.

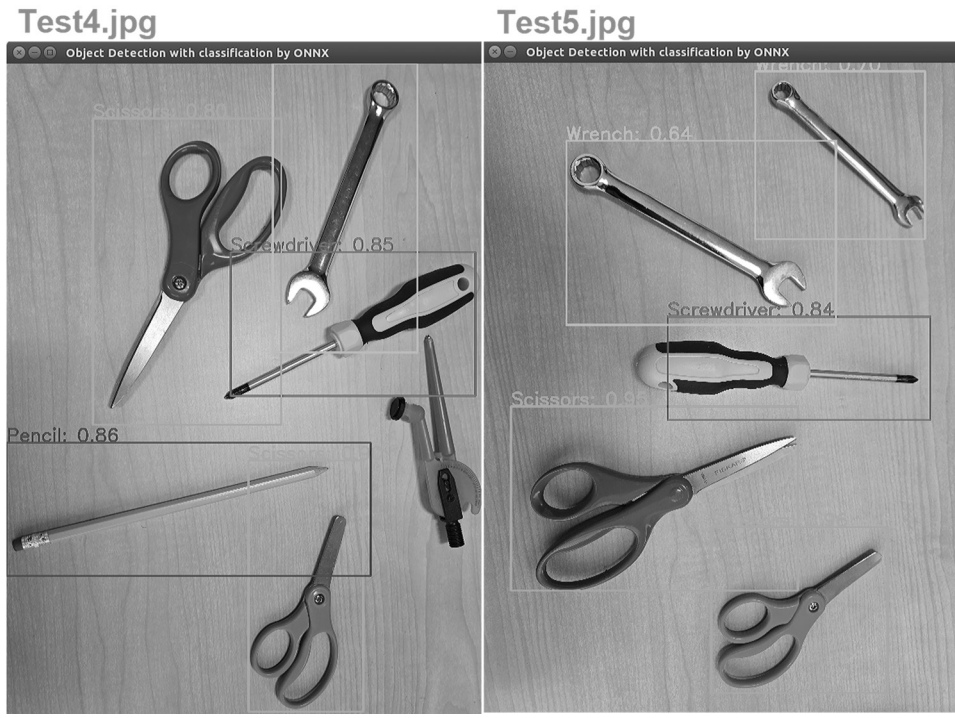


Figure 12.13 Object detection using ONNX model (best_fasterrcnn_model.onnx) in Jetson Nano for images with different sizes.

12.4 Conclusion

This chapter outlines how to convert custom-trained PyTorch models into ONNX format and then use these models for inference on a Jetson Nano. Although PyTorch detection models can be saved as either ".pth" or ".pt" files, the conversion process can differ for object detection models as they can be trained on different backbones. Generally, the ".pt" extension is used when the model saves states, and the ".pth" extension is used to indicate that only the model's parameters are saved without the underlying architecture. This chapter shows steps for both classification and object detection model conversion approaches from a desktop PC to a Jetson Nano along with the processing of inputs and outputs. The approach presented in this chapter (PyTorch model to ONNX) is particularly beneficial when a memory-intensive deep learning model needs to be deployed on portable edge devices such as the Jetson Nano, Raspberry Pi or on specialized hardware.

12.5 Exercise Problems

(Q1) Convert the PyTorch model developed in Chapter 9 (Q1) to ONNX format. Then, transfer the ONNX model to the Jetson Nano and perform real-time inference using a USB video feed.

(Q2) Convert PyTorch's object detection with the classification model developed in Chapter 10 (Q4) to ONNX format. Then, transfer the ONNX model to the Jetson Nano and perform inference on test images.

13

Setting up Raspberry Pi

13.1 Introduction to Raspberry Pi

The Raspberry Pi is a compact single-board computer developed by the Raspberry Pi Foundation in the UK. It was originally developed to promote the teaching of basic computer science in schools. The foundation was formed in 2009, and after several prototypes, the company launched the first model, Raspberry Pi Model B, in 2012. It was sold out almost immediately, marking the beginning of its widespread adoption [1]. Since its first release, it become widely popular beyond its original educational purpose. In the last ten years, the Raspberry Pi Foundation has released several iterations of its board with improved hardware and processing capabilities. Currently, the latest version is Raspberry Pi 5 (released in September 2023), and it represents a significant leap forward in terms of performance, graphics, memory, general purpose input/output (GPIO) pins, and software support compared to its predecessors [2].

Beyond basic computing, the Raspberry Pi is adopting advanced computing applications such as machine learning and deep learning. It supports TensorFlow Lite and PyTorch frameworks for cost-effective edge computing, such as image recognition and object detection. This is particularly useful for developing real-time embedded systems for security purposes and predictive maintenance, where low latency and immediate feedback are necessary. In robotics, the Raspberry Pi serves as a versatile platform for both beginners and advanced users. Its GPIO pins allow integration with a wide range of sensors, motors, and other electronic components [3–5]. Also, its compatibility with popular programming languages Python and C++ provides flexibility in developing custom robotic solutions. It can process data, control movements of motors, and even run complex algorithms for autonomous navigation. For these reasons, Raspberry Pi is one of the popular tools for building and controlling robots, and it can be used in projects ranging from simple remote-controlled cars to advanced drones and robotic arms.

Although the Raspberry Pi's processing power is modest compared to high-end desktop computers or specialized deep learning hardware, it can still run streamlined deep learning models, especially those designed for edge devices. Raspberry Pi's performance can also be enhanced by using hardware accelerators such as the Google Coral USB Accelerator. These features make the Raspberry Pi a cost-effective and practical platform for deploying real-time computer vision applications. In this chapter, we will cover the setup process for a Raspberry Pi 5 and the installation of OpenCV and Pytorch libraries, which are essential for any computer vision applications. In Chapter 14, we will demonstrate how to perform transfer learning for image recognition on a Raspberry Pi board, using the setup described in this chapter.

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

13.2 Hardware and Power Requirements

In this chapter, we demonstrated some common AI capabilities and computer vision programs that can be run on the Raspberry Pi 5. This board offers several improvements over the previous versions, making it a more powerful and versatile platform. It includes a quad-core ARM Cortex-A76 CPU, VideoCore VII GPU, and LPDDR4 RAM (2 GB, 4 GB, or 8 GB), and for storage, it uses a microSD card slot. Raspberry Pi 5 offers a wide range of connectivity options, including Gigabit Ethernet, Wi-Fi, and Bluetooth 5.0. The device has two USB 3.0 ports, and two USB 2.0 ports, and uses USB-C for power input. It also includes two micro-HDMI ports for up to 4 K resolution, two MIPI CSI connectors for camera modules, and MIPI DSI connectors for display modules. Additionally, it has a 40-pin GPIO header for interfacing with various hardware components [6]. We have used the Raspberry Pi 5 (8 GB) with a heatsink (active cooler), as shown in Figure 13.1.

Using a heat sink with a Raspberry Pi is highly recommended to maintain the device's optimal performance. It helps dissipate heat from the processor and other components efficiently, especially when running intensive computations. We have utilized a clip-on active cooler fan to prevent overheating and ensure reliable performance (Figure 13.2). The heatsink's specification can be found at https://www.raspberrypi.com/products/active-cooler, and it has spring-loaded push pins for mounting onto the Raspberry Pi 5 board easily.

The recommended power supply for the Raspberry Pi 5 is a USB-C power adapter that provides high-quality 5 volts and 5 amps (5V/5A). We have used Raspberry Pi 27W USB-C Power Supply available at https://www.raspberrypi.com/products/27w-power-supply. Using a power supply with these specifications is critical, as it ensures that the device receives sufficient power to operate efficiently and support connected peripherals. It also helps avoid problems associated with undervoltage, which can lead to instability, reduced performance, or unexpected shutdowns. Table 13.1 presents all the hardware components that are used in this chapter.

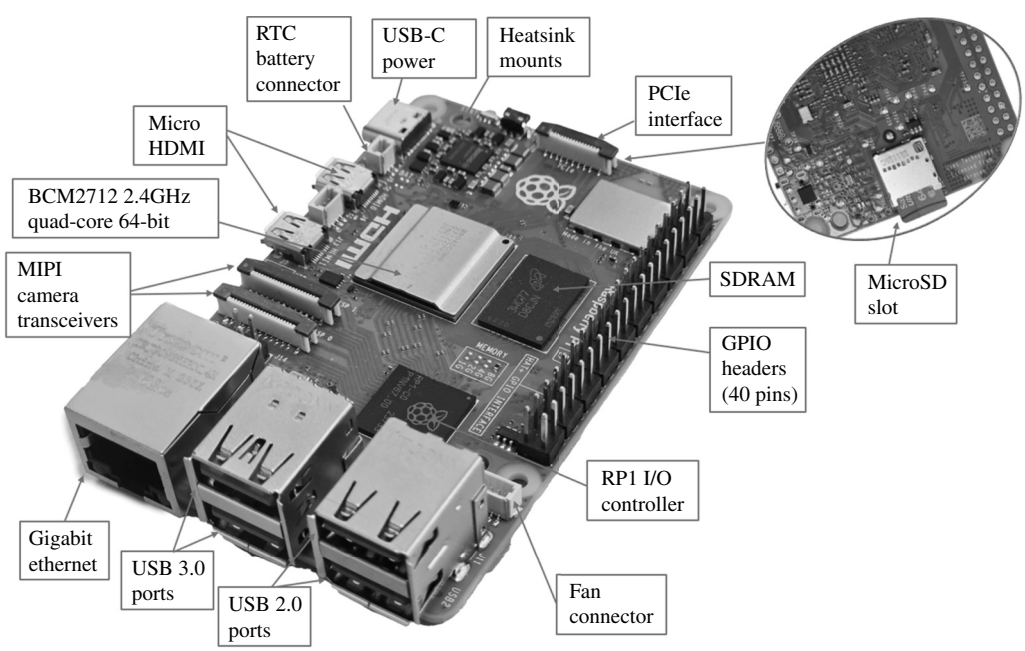


Figure 13.1 The Raspberry Pi 5, along with its key components and peripheral interfaces.

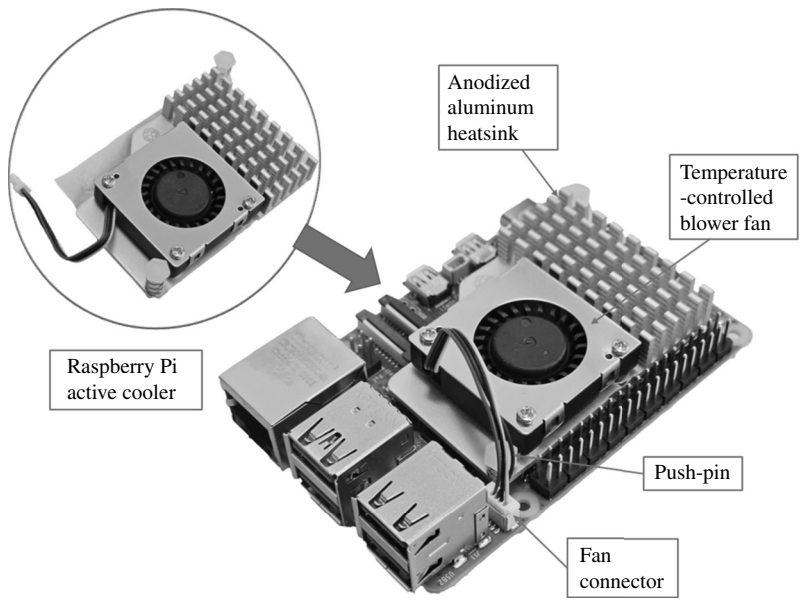


Figure 13.2 Raspberry Pi 5 with heatsink mounted blower fan.

Table 13.1 Hardware components used for Raspberry Pi 5 setup.

Item name	Descriptions	Alternatives
Raspberry Pi 5 board	Raspberry Pi 5 – 8 GB (https://www.sparkfun.com/products/23551?src=raspberrypi)	Other well-known online retailers, such as Amazon, CanaKit, PiShop.us, VILROS, etc., also sell the Raspberry Pi 5 – 8 GB board
Power supply	Raspberry Pi 27W USB-C Power Supply (https://www.sparkfun.com/products/23583?src=raspberrypi)	Other well-known online retailers, such as Amazon, CanaKit, PiShop.us, VILROS, etc., also sell this item. Any other Raspberry Pi 5 compatible 5V/5A power supply will also work
Heatsink	Raspberry Pi Active Cooler (https://www.sparkfun.com/products/23585?src=raspberrypi)	Other well-known online retailers, such as Amazon, CanaKit, PiShop.us, VILROS, etc., also sell this item. Any other Raspberry Pi 5 compatible heatsink with a blower fan will also work
Keyboard	Logitech – K120 Full-size Wired Membrane Keyboard (https://www.logitech.com/en-us/products/keyboards/k120-usb-standard-computer.920-002478.html)	Any other USB keyboard can be used
Mouse	DELL – Optical Mouse MOC5UO USB Wired (https://www.amazon.com/Dell-3-button-Optical-Scroll-XN966/dp/B0029ND59O)	Any other USB mouse can be used

(Continued)

Table 13.1 (Continued)

Item name	Descriptions	Alternatives
Monitor	Lenovo D22e-20 Monitor (https://www.amazon.com/Lenovo-Customizable-Brightness-Flicker-Free-FreeSync/dp/B09QXMHNZ8?th=1)	Any other HDMI monitor can be used
USB camera	Logitech C270 HD Webcam – 720p (https://www.logitech.com/en-us/products/webcams/c270-hd-webcam.960-000694.html)	Although many other similar webcams can be used, this Logitech C270 is highly recommended
HDMI cable	Raspberry Pi Official HDMI Cable (1 m) (https://www.sparkfun.com/products/17387)	Any high-speed male-to-male HDMI cable that supports 4 K resolution will work
Micro HDMI to HDMI	HDMI Adapters Kit (https://www.amazon.com/FUNTEN-Hdmi-Adapters-Micro-Female/dp/B07JFLM2VP?ref_=ast_sto_dp)	Any micro HDMI to HDMI cable should also work. Raspberry Pi's official site listed the following item: https://www.raspberrypi.com/products/micro-hdmi-to-standard-hdmi-a-cable/ This item is also available in various online stores, such as, CanaKit, PiShop.us, VILROS, etc.
MicroSD card	64 GB Extreme microSDXC UHS-I Memory Card with Adapter (https://www.amazon.com/SanDisk-Extreme-microSDXC-Memory-Adapter/dp/B09X7C7LL1)	Any other microSDHC UHS-I with similar MB/s will work
SD card reader	USB 3.0 Portable Card Reader for SD, SDHC, SDXC, MicroSD, MicroSDHC, MicroSDXC (https://www.amazon.com/SmartQ-C307-Portable-MicroSDHC-MicroSDXC/dp/B06ZYXR7DL?th=1)	Other High-Speed USB 3.0 Interfaces will work

13.2.1 Direct Setup

Raspberry Pi 5 will automatically connect to the Wi-Fi network if the network name and password have been set in the image. If not, it can connect to other available Wi-Fi networks using the appropriate password. For the initial setup, using an Ethernet cable is recommended because the first few software packages we will install, such as OpenCV and PyTorch, require a high-speed internet connection. After these installations, Wi-Fi can be used for regular internet access. At this stage, all peripheral connections to the Raspberry Pi board we used are shown in Figure 13.3.

13.3 Operating System Setup

The Raspberry Pi uses a Debian-based operating system called Raspberry Pi OS, which was previously known as Raspbian. This is chosen due to its lightweight design, user-friendliness, and excellent compatibility with the Raspberry Pi hardware. This OS is optimized for the ARM

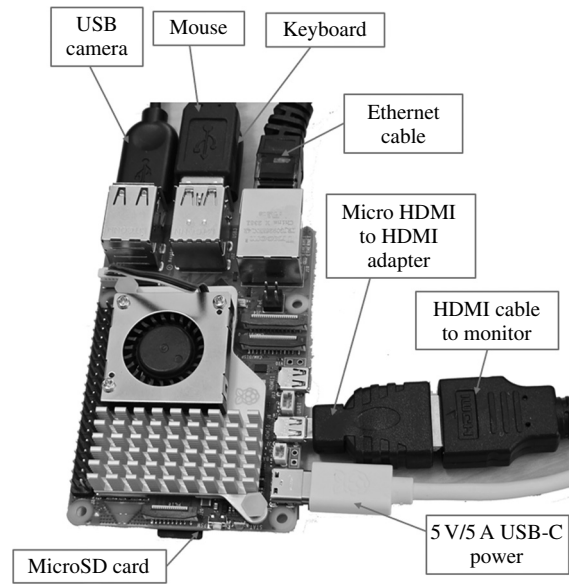


Figure 13.3 Peripheral connections to the Raspberry Pi 5 board with 5V/5A power supply.

architecture that Raspberry Pi uses, and it provides a robust and efficient platform for various applications. It is also pre-loaded with a wide variety of software and tools specifically chosen for educational purposes, development work, and DIY projects.

13.3.1 Flashing the OS Image

The latest version of Raspberry Pi OS is built on Debian 12 and is known as "Bookworm" [7]. It's the operating system of choice for Raspberry Pi 5, designed to harness the latest software features, security updates, and performance improvements by Debian 12. To install "Bookworm" on a Raspberry Pi 5, we first need to flash the appropriate image onto a microSD card. For this purpose, we used a 64 GB microSDXC UHS-I card. The microSD card was connected to a Windows computer via a SmartQ C307 USB 3.0 Portable Card Reader and formatted using an SD card formatter application, which can be downloaded from (https://www.sdcard.org/downloads/formatter/sd-memory-card-formatter-for-windows-download). Make sure to format the correct drive and verify available disk space while using the formatter application, as shown in Figure 13.4.

There are several ways to flash the microSD with an operating system. We have used an application called Raspberry Pi Imager, which is a simple, user-friendly tool developed by the Raspberry Pi Foundation to streamline the process of installing operating systems on Raspberry Pi devices. The Imager supports a variety of operating systems, including Raspberry Pi OS, Ubuntu, and other specialized distributions, and it can be downloaded from Raspberry Pi's website (https://www.raspberrypi.com/software). After downloading the Imager (Figure 13.5), we can install it on our Windows system by double-clicking the installer and following the straightforward setup process.

Launch the Raspberry Pi Imager application and select a device. Since we are using Raspberry Pi 5 in this chapter, we chose Raspberry Pi 5, as shown in Figure 13.6.

To select the "Bookworm" operating system, click on "Choose OS" in the Imager and then select "Raspberry Pi OS (other)." Next, choose "Raspberry Pi OS Full (64 bit)," which includes the desktop environment and recommended applications, as shown in Figure 13.7.

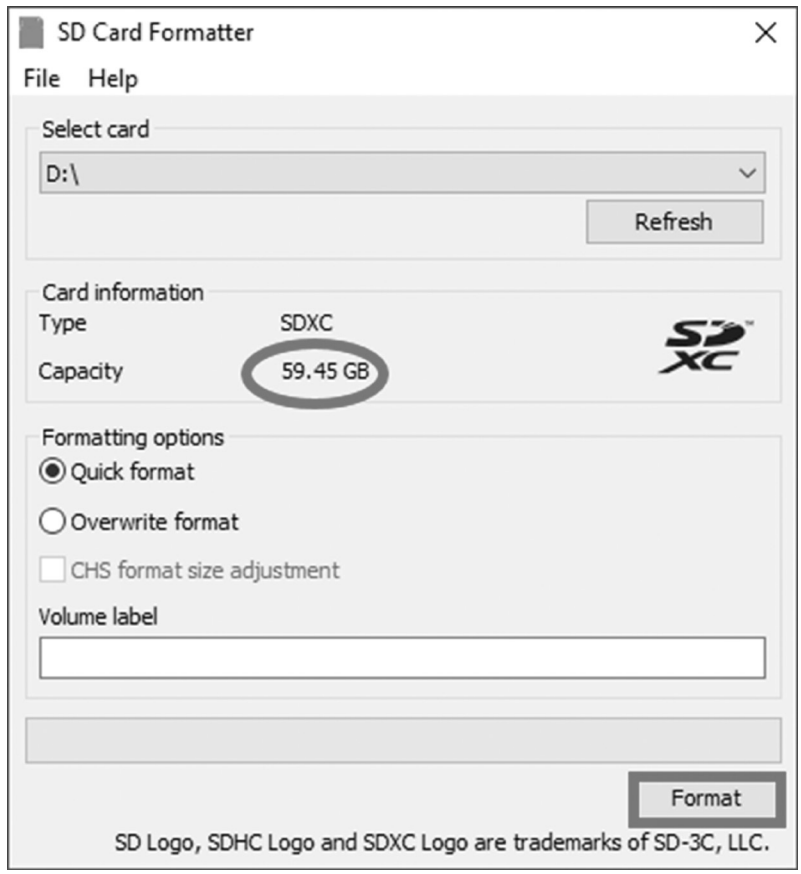


Figure 13.4 Format 64 GB microSDXC card using SD card formatter application.

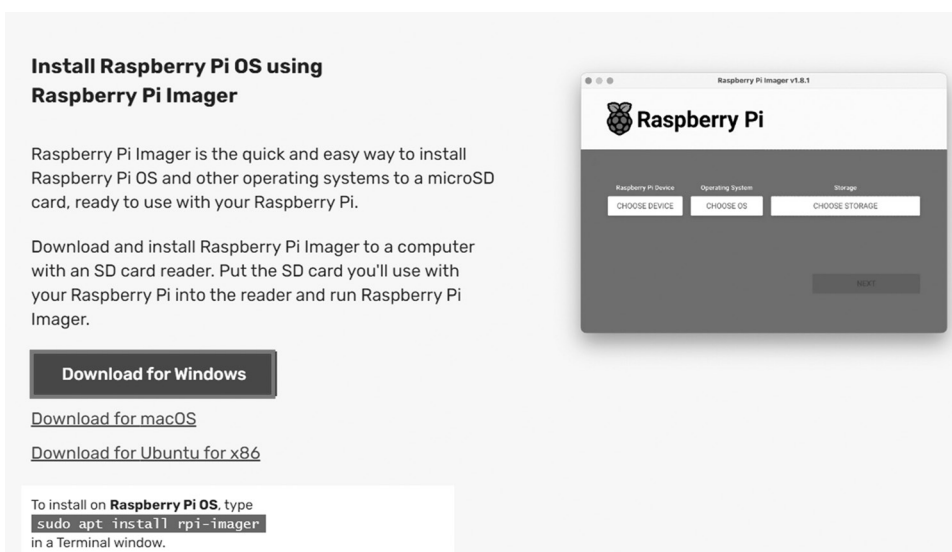


Figure 13.5 Download Raspberry Pi imager for Windows computer.

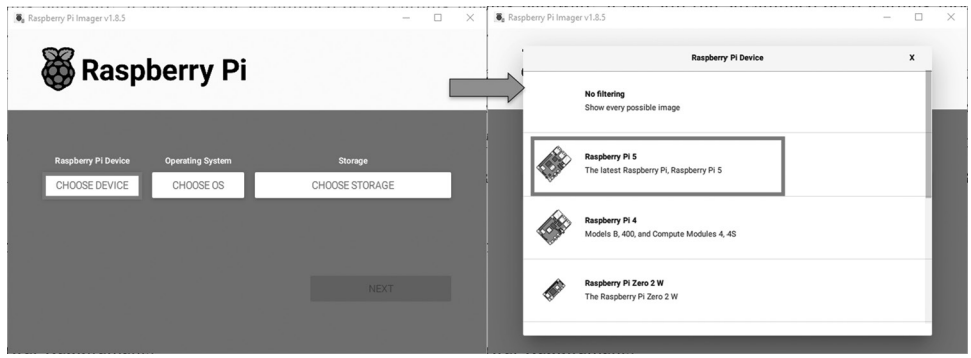


Figure 13.6 Select "Choose Device" from the Raspberry Pi Imager and "Raspberry Pi 5" device.

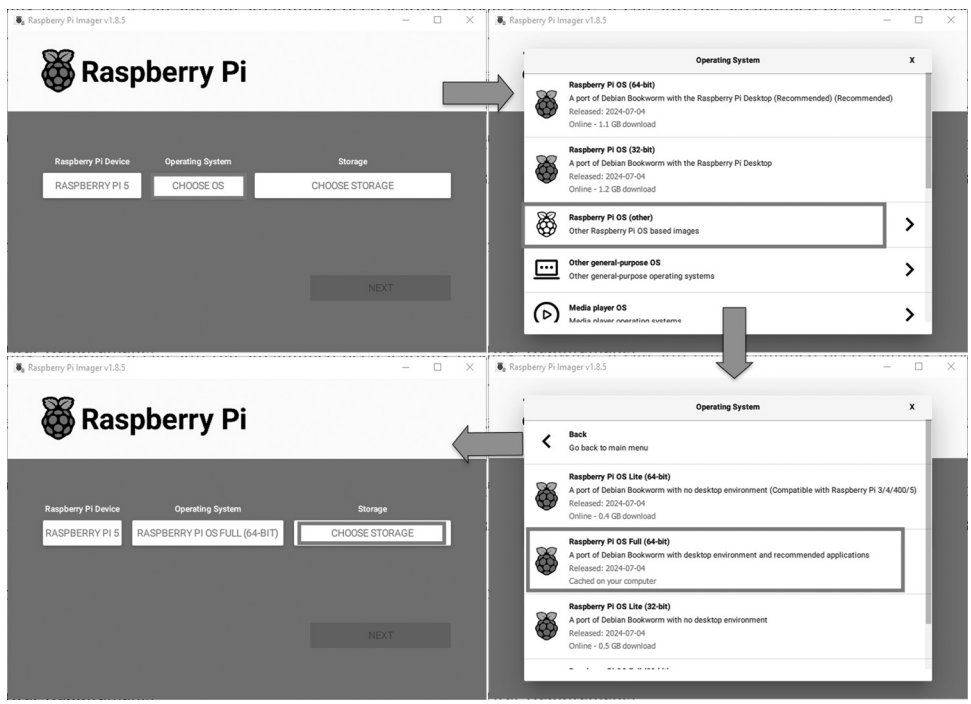


Figure 13.7 Select "Choose OS" from Raspberry Pi Imager and navigate to select the "Raspberry Pi OS Full (64-bit)" Debian Bookworm operating system.

Next, click on the "Choose Storage" option in the Imager and select the "SD Card Reader USB Device" for flashing the operating system. Then, click "Next," and in the following window, select "Edit Settings." In the "Edit Settings" window, you can set a username and password for the Raspberry Pi operating system. In the example shown in Figure 13.8, we chose "tma" as the username and a one-word password. Although a one-word password is not recommended, it is convenient during installation and other operations when the password needs to be entered multiple times, especially if only one person is testing the device. For the final project deployment, a stronger password should be used to enhance security.

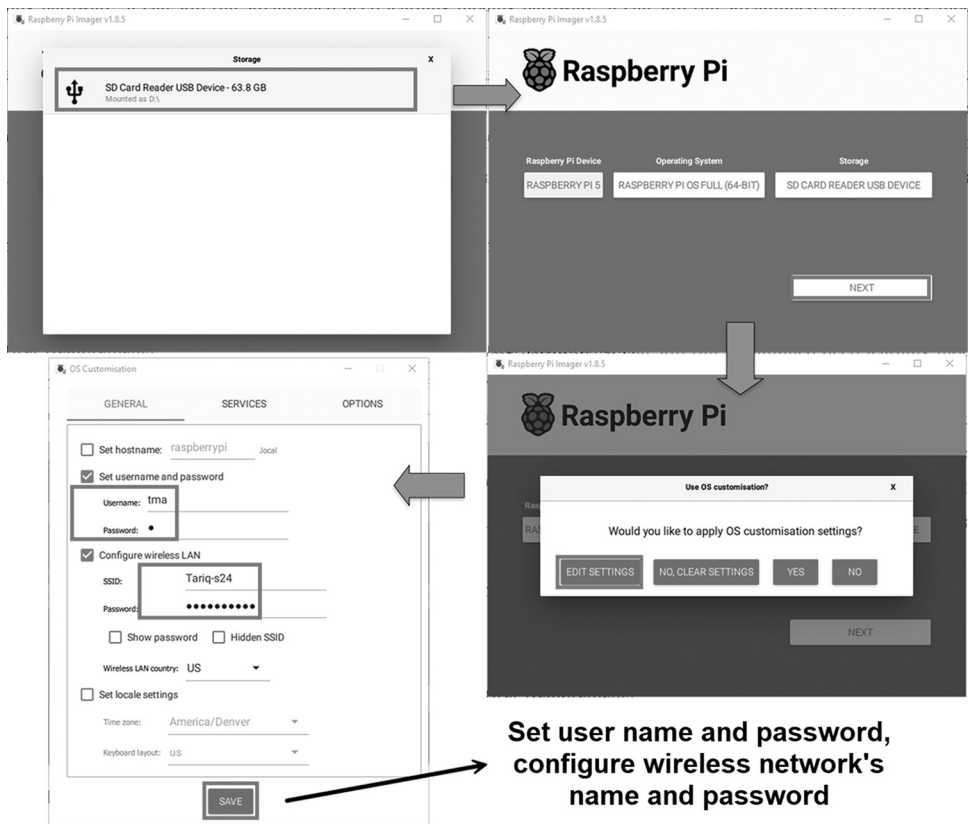


Figure 13.8 Select the SD card reader USB device (microSD) and configure the OS customization settings to set up the username and password, as well as the wireless network name and password.

In the "Edit Settings" window, we can also configure the wireless network name and password for automatic connection. This is straightforward for private or home networks. However, connecting to a secure school network such as "eduroam" with two-factor authentication is not convenient. To simplify the process, users can connect to the "eduroam" network via a smartphone and create a "Mobile Hotspot and Tethering" username and password on the phone. We used a Samsung S24+ smartphone to create the username "Tariq-s24" and a password, then used this information to configure the wireless LAN (Figure 13.9).

After saving the "Edit Settings," apply the OS customization settings by selecting "Yes," and then confirm the warning that existing data on the SD card reader USB device will be erased by selecting "Yes" again (Figure 13.9). Raspberry Pi Imager will write and then verify the image. During or after verification, multiple windows may pop up asking to format the disk (depending on Windows security settings). Cancel all these windows and then remove the SD card reader USB device.

13.3.2 Setting up OS

Setting up the Raspberry Pi 5 is quite simple. Initially, we need to connect all the peripherals to the Raspberry Pi 5, as illustrated in Figure 13.3. Alternatively, we may also remote login to the device using secure shell (SSH), real virtual network computing (VNC), or Putty software. To keep things straightforward, we opted for a direct connection with a keyboard, mouse, monitor, USB camera,

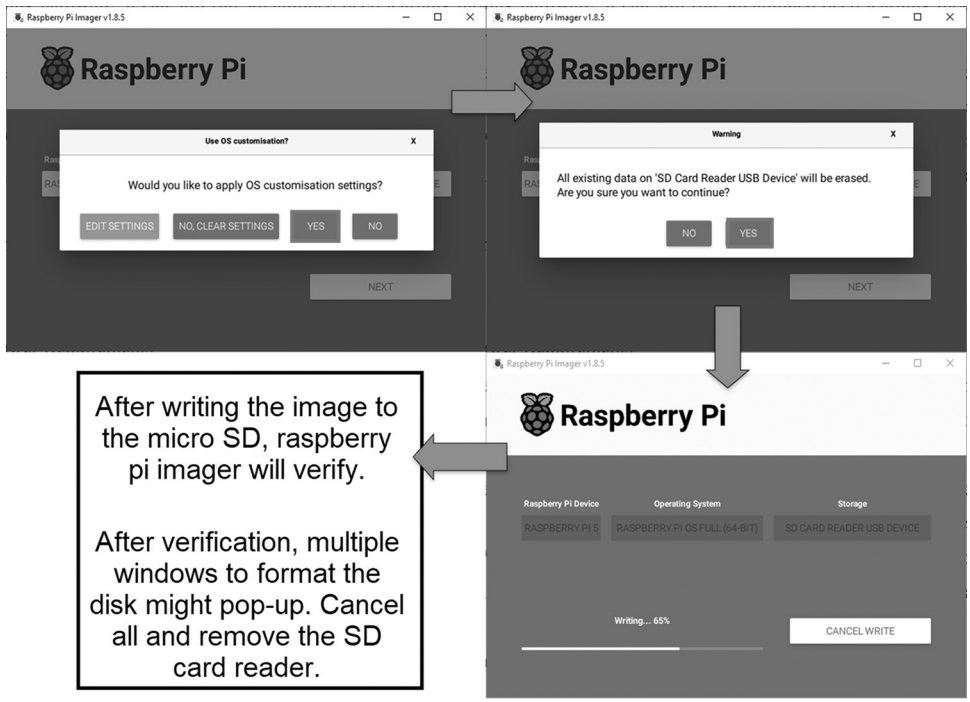


Figure 13.9 Apply OS customization settings and select "Yes" to proceed with writing and verification.

and Ethernet cable. Upon the first boot, a screen similar to Figure 13.10 will appear, and we can verify that Wi-Fi is already available and connected to the network assigned during the image flashing process. While Wi-Fi can be used for some setup tasks and browsing, it is strongly recommended to use an Ethernet cable for the initial setup, especially when installing OpenCV and PyTorch.

If the Wi-Fi network and username are not configured during the flashing of the operating system image, as depicted in Figure 13.8, a blue screen will appear, prompting us to set the keyboard layout, username, and password. Once these details are entered, we will be able to log in to the Raspberry Pi. The initial setup will then proceed to a page where we can select the Wi-Fi network. In our case, this step is not required.

Now, navigate to the "Mouse and Keyboard settings" through the Raspberry Pi Menu "Preferences," choose the "Keyboard" tab, and set the keyboard layout and variant to "English (US)." Confirm the selection by pressing OK. On different keyboard layouts, certain special characters (e.g. ~) needed in the command prompt might be hard to locate. Next, go to "Preferences," select "Raspberry Pi Configuration," and open the "Interfaces" tab. Here, enable the "Raspberry Pi Connect" option. These steps are shown in Figure 13.11.

Next, open the Raspberry Pi terminal window using the "terminal" icon from the desktop, and update and upgrade using the following commands:

$ sudo apt-get update
$ sudo apt-get upgrade

These commands are essential for maintaining and upgrading the system software on a Raspberry Pi, as well as other Debian-based Linux distributions. The "sudo apt-get update" command refreshes the local package indices and synchronizes those with the latest software versions available in the repositories. This step doesn't install or upgrade anything. On the other hand, "sudo apt-get upgrade" is used

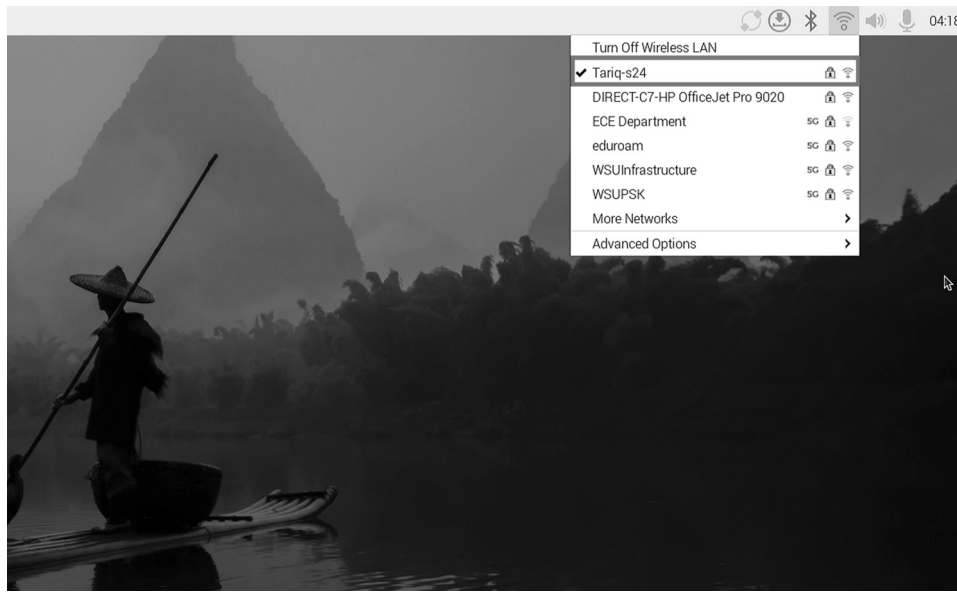


Figure 13.10 Verify the Wi-Fi network and internet connection in the Raspberry Pi.

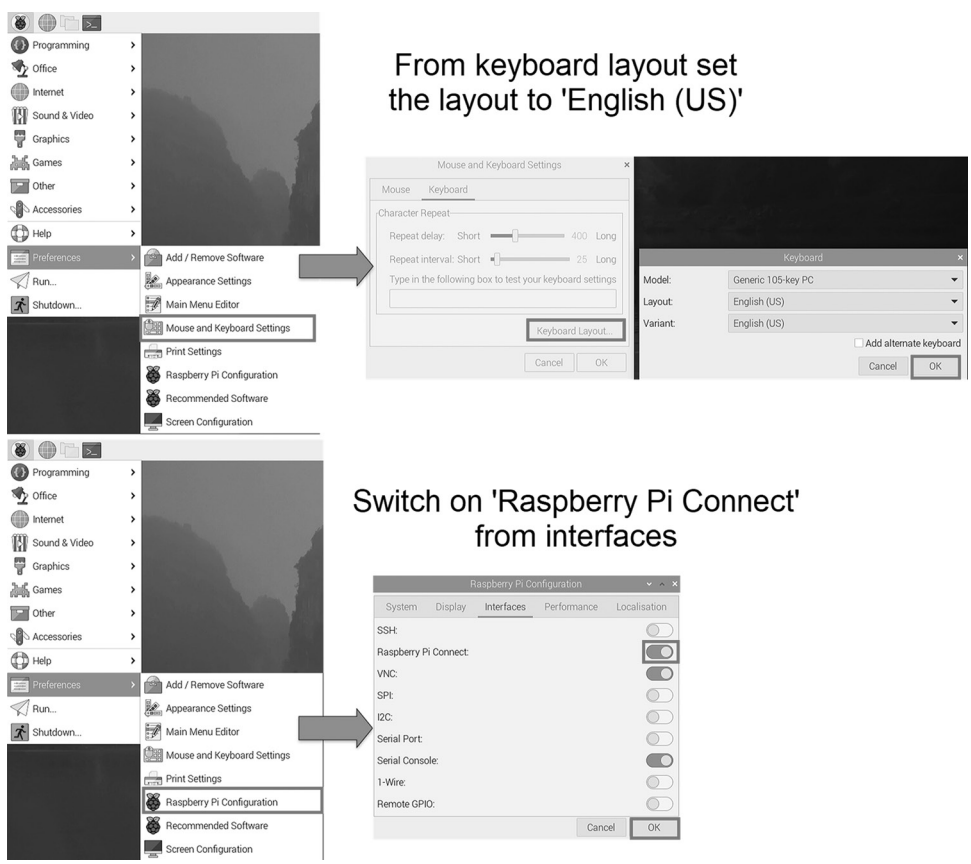


Figure 13.11 Set the Keyboard layout to English (US) and switch on the "Raspberry Pi Connect" option.

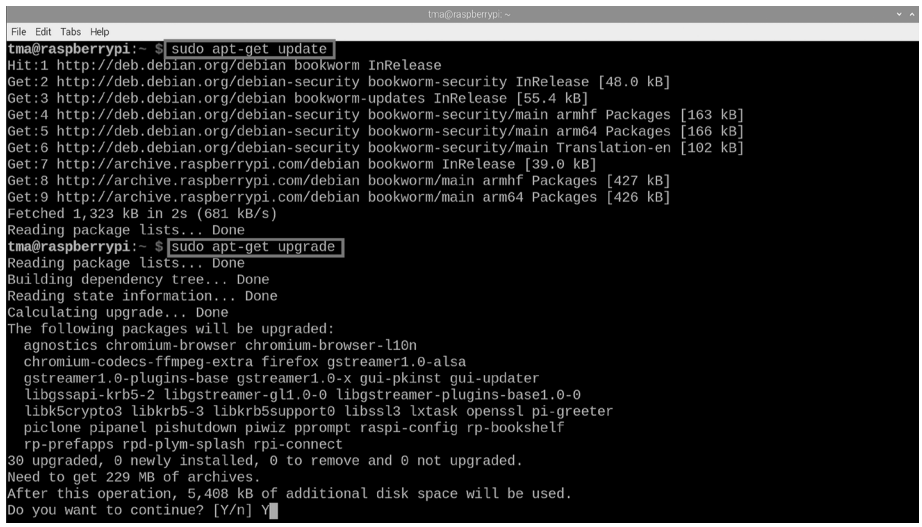


Figure 13.12 Update and upgrade the system software and Linux distributions.

to install the newest versions of all packages currently on the system. This command also handles any required dependency changes, but it won't remove existing packages. Figure 13.12 shows the execution of these two commands.

13.4 Create Virtual Environment

Our next step would be creating a virtual environment, as it is highly beneficial for managing Python projects that use OpenCV and PyTorch. It creates separate, isolated spaces for a set of similar projects and allows handling dependencies independently without impacting the global Python installation. For example, if we install TensorFlow, it will also install NumPy as a dependency. Later, if we install PyTorch, the system might need a newer NumPy version, and we could run into compatibility problems with the TensorFlow setup. By using separate virtual environments, we can keep the NumPy version intact in one environment while using a different NumPy version in another. This approach helps maintain the stability of various Python projects and avoids version conflicts.

First, verify the Python3 version and its location, then add this location to the "~/.bashrc" file and reload it using the following commands:

$ which python 3.9
$ echo "export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python" >> ~/.bashrc
$ source ~/.bashrc

To find the correct Python version, type python3 in the terminal and press Enter. This will start Python3 and display the version at the beginning. To return to the terminal, press "Ctrl+d". Install the virtual environment using the following commands:

$ sudo apt-get install python3-virtualenv
$ sudo apt-get install python3-virtualenvwrapper

The virtual environment wrapper will extend the functionality of virtual environments and make them easier to manage. These steps are illustrated in Figure 13.13.

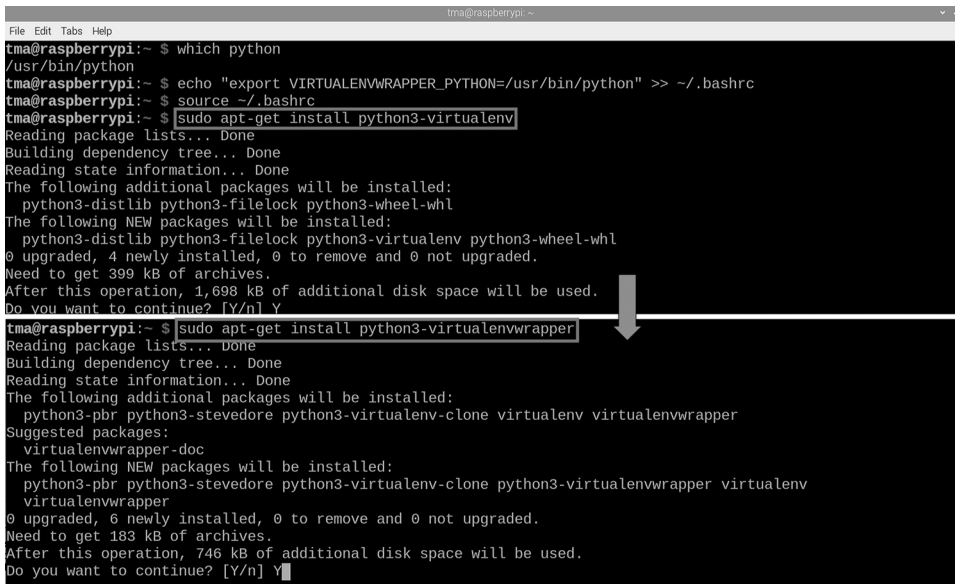


Figure 13.13 Install the Python3 virtual environment and virtual environment wrapper from the terminal.

After these installations, run the following commands to configure the virtual environment wrapper and update "~/.bashrc" file:

$ echo "export WORKON_HOME=$HOME/.virtualenvs" >> ~/.bashrc
$ echo "source /usr/share/virtualenvwrapper/virtualenvwrapper.sh" >> ~/.bashrc
$ source ~/.bashrc

Next, we would like to create a virtual environment for the AI tasks, named "mydl." It will be our self-contained directory that includes a Python interpreter and a specific set of libraries. Use the following commands to create the environment and deactivate it:

$ mkvirtualenv mydl
$ deactivate

To work on the environment again, enter "workon mydl" in the terminal, as shown in Figure 13.14.

13.5 PyTorch and OpenCV Installation

PyTorch is the popular open-source deep learning framework developed by Facebook's AI Research lab [8]. It is very useful for edge computing, where AI tasks such as image recognition, object detection, and data analysis need to be performed locally using a pre-trained network. By using PyTorch on Raspberry Pi, developers can prototype and implement AI solutions in a cost-effective and scalable manner. To install the Pytorch, first install some essential packages such as "pip" installer, "libjpeg-dev" package for JPEG image processing, "libopenblas-dev" for OpenBLAS (Basic Linear Algebra Subprograms) high-performance linear algebra operations, and "libopenmpi-dev" for open multiprocessing library. To install the libraries, use the following commands:

$ sudo apt-get install python3-pip libjpeg-dev libopenblas-dev libopenmpi-dev libomp-dev

```
tma@raspberrypi:~ $ echo "export WORKON_HOME=$HOME/.virtualenvs" >> ~/.bashrc
tma@raspberrypi:~ $ echo "source /usr/share/virtualenvwrapper/virtualenvwrapper.sh" >> ~/.bashrc
tma@raspberrypi:~ $ source ~/.bashrc
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/premkproject
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/postmkproject
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/initialize
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/premkvirtualenv
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/postmkvirtualenv
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/prermvirtualenv
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/postrmvirtualenv
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/predeactivate
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/postdeactivate
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/preactivate
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/postactivate
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/get_env_details
tma@raspberrypi:~ $ mkvirtualenv mydl
created virtual environment CPython3.11.2.final.0-64 in 221ms
  creator CPython3Posix(dest=/home/tma/.virtualenvs/mydl, clear=False, no_vcs_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=/home/
tma/.local/share/virtualenv)
    added seed packages: pip==23.0.1, setuptools==66.1.1, wheel==0.38.4
  activators BashActivator,CShellActivator,FishActivator,NushellActivator,PowerShellActivator,PythonActivator
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/mydl/bin/predeactivate
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/mydl/bin/postdeactivate
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/mydl/bin/preactivate
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/mydl/bin/postactivate
virtualenvwrapper.user_scripts creating /home/tma/.virtualenvs/mydl/bin/get_env_details
(mydl) tma@raspberrypi:~ $ deactivate
tma@raspberrypi:~ $ workon mydl
(mydl) tma@raspberrypi:~ $
```

Figure 13.14 Update the "~/.bashrc" file and create a virtual environment called "mydl." To exit the virtual environment, use the "deactivate" command, and to enter the virtual environment, use the "workon mydl" command.

After these installations, enter the "mydl" virtual environment using the "workon mydl" command. Inside the "mydl" environment, install "numpy cython," "requests," "pytorch," "torchvision," and "torchaudio" using the commands below:

$ workon mydl
$ pip3 install setuptools numpy Cython
$ pip3 install requests
$ pip3 install torch torchvision --index-url https://download.pytorch.org/whl/cpu
$ pip3 install torchaudio --index-url https://download.pytorch.org/whl/cpu

Make sure to be inside the "mydl" environment while performing these installations. These steps are illustrated in Figure 13.15. "Cython" allows developers to write Python code that is compiled into C for performance optimization, and "requests" is a user-friendly library for making HTTP requests. The "torchvision" and "torchaudio" libraries are extensions of the PyTorch framework to provide utilities and models for computer vision and audio tasks. These libraries can make Raspberry Pi a compact platform for prototyping and deploying various types of AI projects.

Next, we will need to install OpenCV, which is the popular open-source library designed for real-time computer vision and image processing [9]. OpenCV is highly useful for Raspberry Pi due to its ability to efficiently perform complex image processing tasks on limited hardware resources. To install OpenCV use the following command:

$ pip3 install opencv-contrib-python

Ensure that you are within the "mydl" virtual environment while performing this installation (Figure 13.16). To verify if the installation was successful, enter Python3 from the "mydl" environment and import and check the versions of "torch" and "cv2." If the installation is not done correctly, you will encounter an error at this point. Figure 13.16 shows these steps.

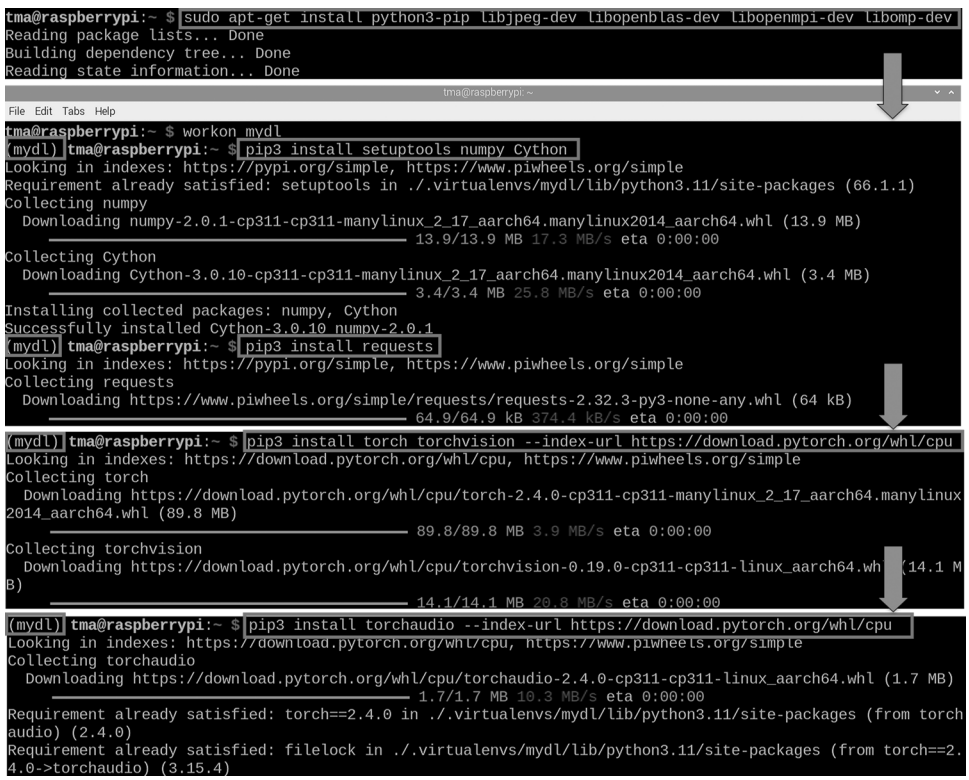


Figure 13.15 Install "pip," "libjpeg-dev," "libopenblas-dev," and "libopenmpi-dev" libraries; enter "mydl" virtual environment, and then install "cython," "requests," "pytorch," "torchvision," and "torchaudio."

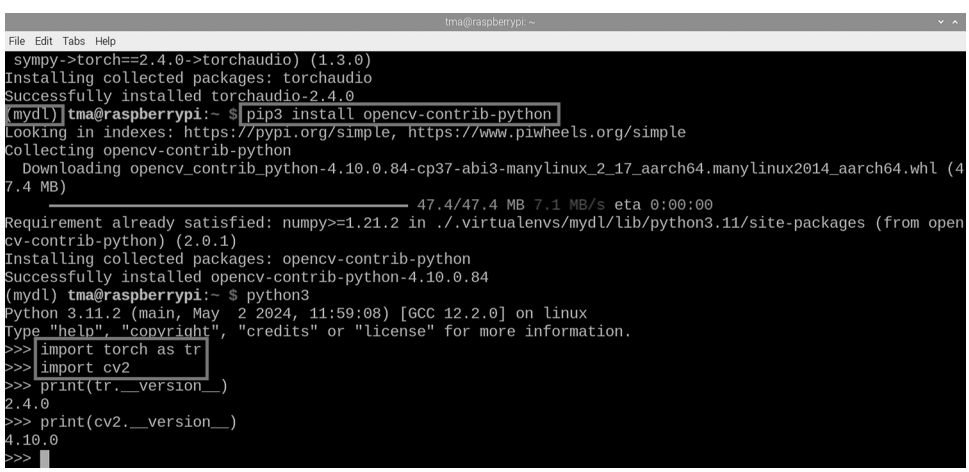


Figure 13.16 Install OpenCV in the virtual environment and confirm the installation of both PyTorch and OpenCV using Python3.

13.6 Other Essential Packages

We will now install Pillow (PIL), which is a popular Python Imaging Library (PIL) fork for flexible image processing and manipulation operation. It offers a wide range of features, such as accessing, modifying, and storing different image file types, along with fundamental image tasks such as adjusting size, trimming, and shifting between colors [10]. Pillow is particularly suitable for use with a Raspberry Pi due to its lightweight nature and efficiency. From the "mydl" virtual environment, upgrade "pip" and install "pillow" using the following commands (Figure 13.17).

$ python -m pip install --upgrade pip
$ python -m pip install --upgrade Pillow

Next, we are going to install "onnx" and "onnxscript." These installations will enable comprehensive support for working with the open neural network exchange (ONNX) format, which facilitates interoperability between different deep learning frameworks [11]. The "onnx" toolkit offers functionality to transform models from different frameworks into ONNX format, and "onnxscript" enables developers to write ONNX models using scripting language. To install these, use the following commands, as shown in Figure 13.18.

$ pip install onnx
$ pip install onnxscript

Next, to add more packages to the virtual environment, we require the "cmake" tool for compiling and building software from the source code. Additionally, we can now install the GTK+3 toolkit for creating graphical user interfaces. Use the following terminal commands (outside the virtual environment) for a system-wide installation (Figure 13.19).

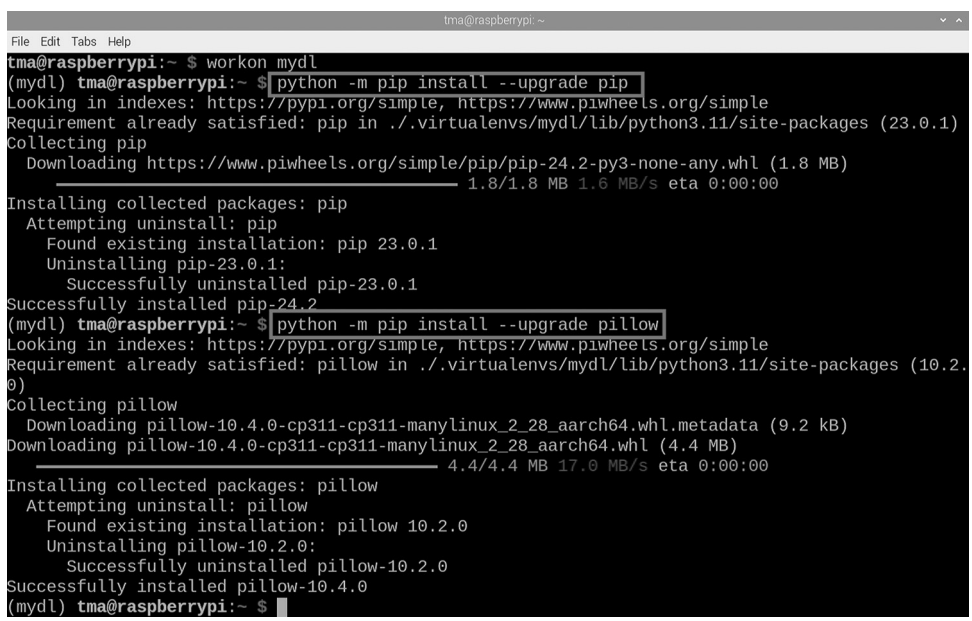


Figure 13.17 Install and upgrade "pip" and "pillow."

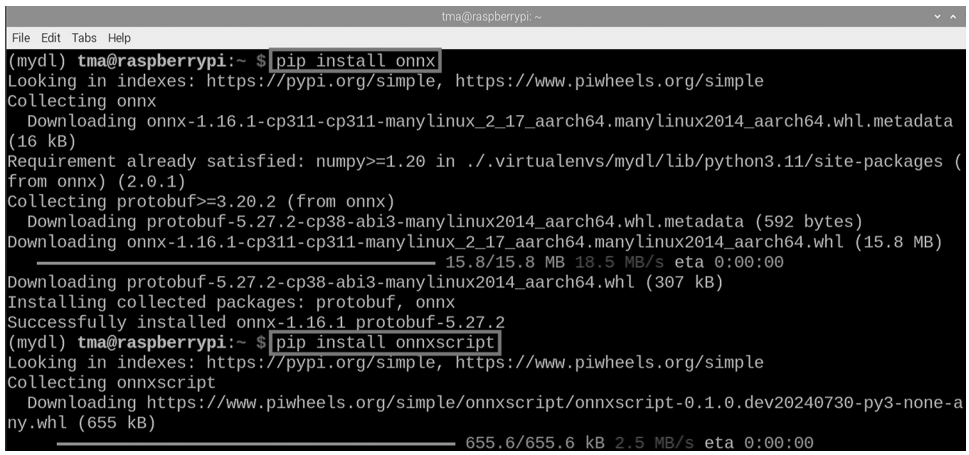


Figure 13.18 Install "onnx" and "onnxscript" from the "mydl" virtual environment.

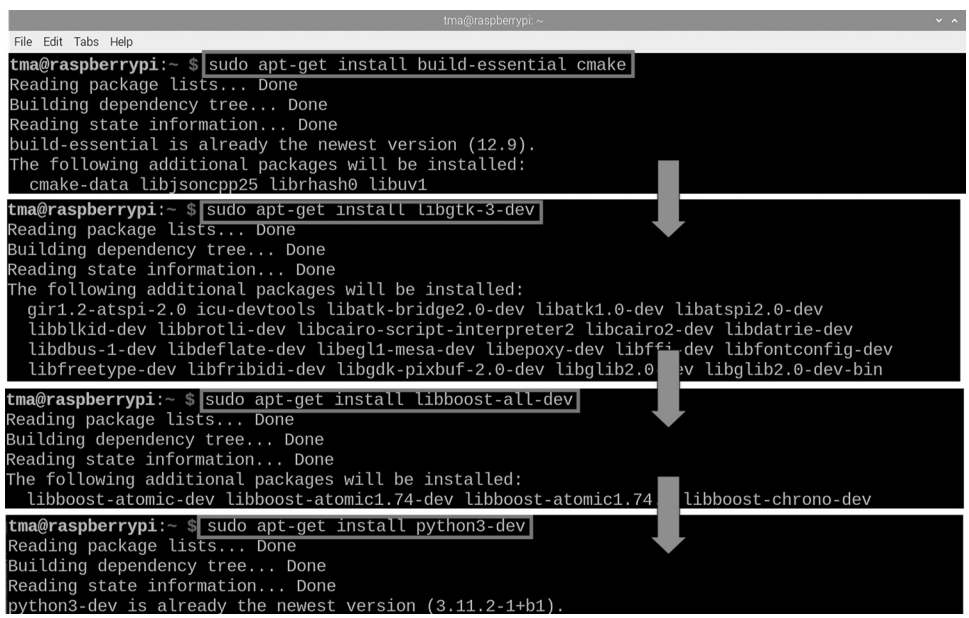


Figure 13.19 Install "cmake," "libgtk-3-dev," "libboost-all-dev," and "python3-dev" from the terminal.

$ sudo apt-get install build-essential cmake
$ sudo apt-get install libgtk-3-dev
$ sudo apt-get install libboost-all-dev
$sudo apt-get install python3-dev

After these installations, go to the "mydl" virtual environment and install "wheel," "dlib," and "face-recognition," as shown in Figure 13.20. Although "wheel" is likely already installed from a previous package installation, we can confirm it using the "pip install wheel" command. We will need the "wheel" package to streamline software distribution and installation and the "dlib" library for image processing and computer vision [12]. The "face-recognition" library is built on "dlib" and

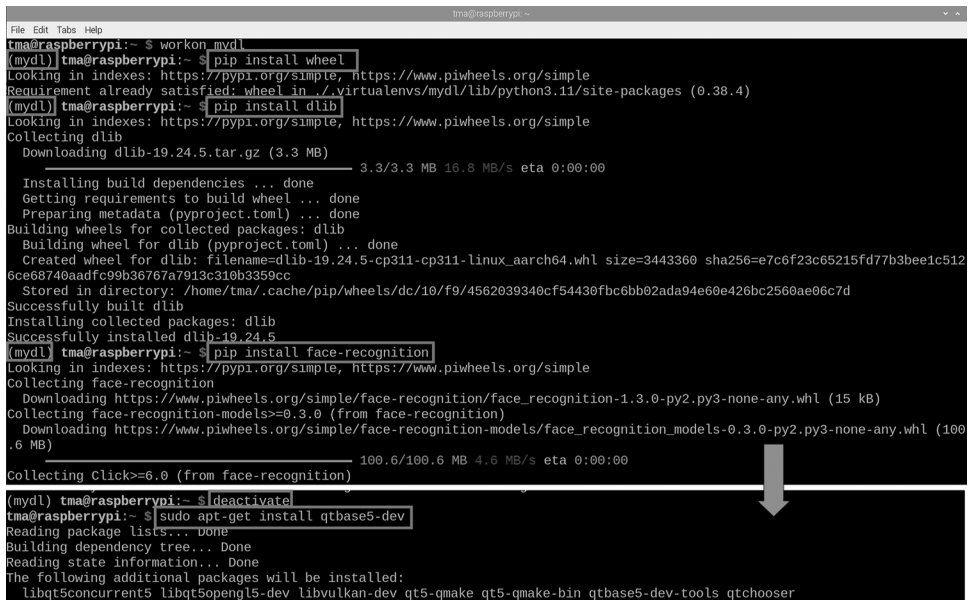


Figure 13.20 Install "wheel," "dlib," and "face-recognition" in the "mydl" virtual environment and install "qtbase5-dev" globally by deactivating the virtual environment.

provides user-friendly and robust tools for facial detection, recognition, and manipulation in Python applications [13]. The "qtbase5-dev" development package for Qt5 will be helpful for creating GUI applications using the Qt framework, and it needs to be installed globally outside of the virtual environment. To exit the virtual environment, type and enter "deactivate," and then install "qtbase5-dev" using sudo. Commands for installing these packages and libraries are given below:

$ pip install wheel
$ pip install dlib
$ pip install face-recognition
$ sudo apt-get install qtbase5-dev

Another step we need to take is to downgrade the current "numpy" version to ensure compatibility with the "dlib" library. There are some existing issues with NumPy 2.0 that cause problems with "dlib." This step may become unnecessary once "numpy" addresses and fixes this issue. To uninstall NumPy 2.0.1 and install NumPy 1.26.4, use the following commands within the virtual environment (Figure 13.21). Before proceeding with the downgrade, we may verify the current version of NumPy by importing "numpy" in Python3.

$ pip uninstall numpy
$ pip install numpy==1.26.4

While running the example codes from this chapter, we may encounter errors related to Wayland backends. This should not significantly impact the overall performance of the Raspberry Pi. However, if desired, we can update this configuration. To do so, enter the command "sudo raspi-config" in the terminal, navigate to "Advanced Options," and switch between "X" and "Wayland backends." The primary differences between X and Wayland backends are in their structure, efficiency, and graphical rendering methods.

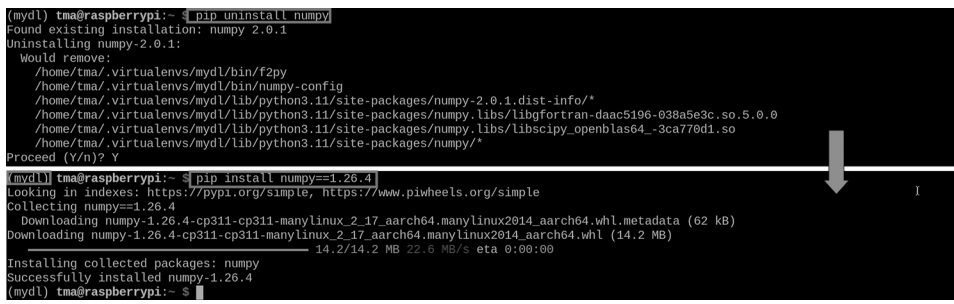


Figure 13.21 Downgrade "numpy" library from version 2.0.1 to 1.26.4.

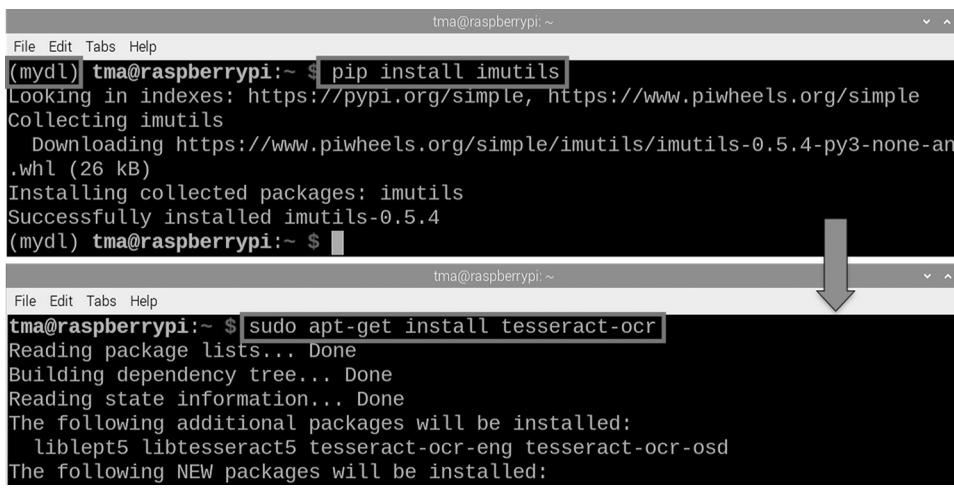


Figure 13.22 Install "imutils" within the "mydl" virtual environment and "tesseract-ocr" globally from the terminal.

The "imutils" offers helpful functions for fundamental image processing tasks such as resizing, rotating, translating, and displaying images, which can be used with OpenCV. The "tesseract" is an open-source Optical Character Recognition (OCR) engine that transforms images of text into machine-encoded text. The commands for these installations are provided below:

$ pip install imutils
$ sudo apt-get install tesseract-ocr

Both of these libraries are useful for Raspberry Pi, as they can efficiently handle image processing and text recognition tasks. The "imutils" library is installed within the virtual environment, whereas "tesseract" is installed globally from the terminal. Installing "tesseract" within the virtual environment might lead to issues such as missing files or inability to locate files when running a text recognition program. Figure 13.22 illustrates the installation process for these libraries.

Now we need to set an environment variable that tells Qt applications to use the X11 window system. The command for setting up this environment is given below:

$ export QT_QPA_PLATFORM=xcb

The Qt XCB platform setup on a Raspberry Pi enables the development and execution of graphical applications using the Qt framework and the X Window System. After that, enter the virtual environment "mydl," and install tesseract and "pytesseract" using the following commands.

$ pip install tesseract
$ pip install pytesseract

The "xcb" setup from the global environment, and the "tesseract" and "pytesseract" installation processes from the virtual environment are illustrated in Figure 13.23. The "pytesseract" is a Python wrapper for "tesseract," and it provides an easy-to-use interface to access Tesseract's optical character recognition (OCR) capabilities directly from Python scripts [14]. These two tools allow developers to incorporate text recognition capabilities into their applications and enable the automation of extracting information from images.

The "xcb" environment setting is temporary and will need to be reapplied if the system is restarted. To make this setting permanent, we can modify the "~/.bashrc" shell script so that is set for every new bash session.

Enter the following command in the terminal:

$ nano ~/.bashrc
 # add the following line at the end and save the file by pressing "ctrl+x," "y," and enter
export QT_QPA_PLATFORM=xcb

This procedure is shown in Figure 13.24. After adding "*export QT_QPA_PLATFORM=xcb*" at the end of the "~/.bashrc" file, save it by pressing "ctrl+x," "y," and enter. Then reboot the system.

```
tma@raspberrypi:~ $ export QT_QPA_PLATFORM=xcb
tma@raspberrypi:~ $ workon mydl
(mydl) tma@raspberrypi:~ $ pip install tesseract
Looking in indexes: https://pypi.org/simple, https://www.piwheels.org/simple
Collecting tesseract
  Downloading https://www.piwheels.org/simple/tesseract/tesseract-0.1.3-py3-none-any.whl (45.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 45.6/45.6 MB 3.3 MB/s eta 0:00:00
Installing collected packages: tesseract
Successfully installed tesseract-0.1.3
(mydl) tma@raspberrypi:~ $ pip install pytesseract
Looking in indexes: https://pypi.org/simple, https://www.piwheels.org/simple
Collecting pytesseract
  Downloading https://www.piwheels.org/simple/pytesseract/pytesseract-0.3.10-py3-none-any.whl (14 kB)
Collecting packaging>=21.3 (from pytesseract)
  Downloading https://www.piwheels.org/simple/packaging/packaging-24.1-py3-none-any.whl (53 kB)
Requirement already satisfied: Pillow>=8.0.0 in ./.virtualenvs/mydl/lib/python3.11/site-packages (from p
ytesseract) (10.4.0)
Installing collected packages: packaging, pytesseract
Successfully installed packaging-24.1 pytesseract-0.3.10
(mydl) tma@raspberrypi:~ $
```

Figure 13.23 Set Qt "xcb" platform globally, and install "tesseract" and "pytesseract" from the virtual environment.

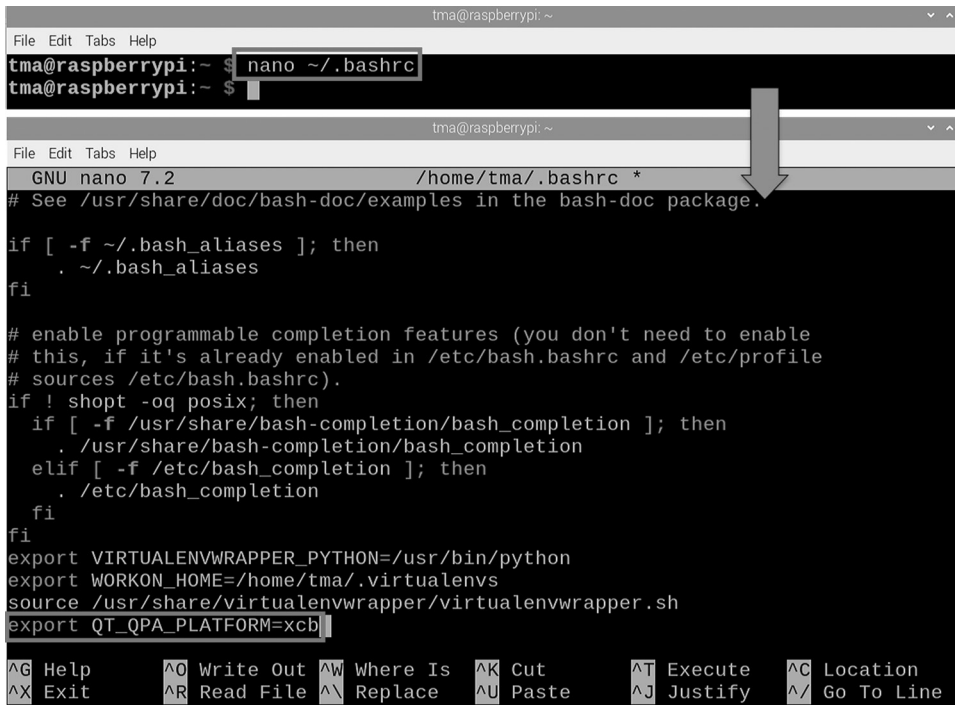


Figure 13.24 Add "export QT_QPA_PLATFORM=xcb" at the end of the "~/.bashrc" file and save it by pressing "ctrl+x," "y," and enter.

13.7 Conclusion

This chapter illustrates the process of configuring a Raspberry Pi 5 for machine and deep learning applications. As shown in Sections 13.4 and 13.5, installing OpenCV and PyTorch within a virtual environment is crucial on a Raspberry Pi, since doing so globally can lead to compatibility problems. Together, these two packages create a powerful environment for deploying deep learning applications on the Raspberry Pi 5 and can make it an ideal platform for low-cost and portable edge AI applications.

Raspberry Pi is more appropriate for transfer learning than full-scale deep learning training due to its computational constraints. Its CPU and GPU cannot match the power of high-end desktops or specialized servers with substantial processing capabilities and memory. However, transfer learning, which involves adjusting and deploying pre-trained models, is less resource-intensive and can be performed on the Raspberry Pi. This makes it suitable for tasks such as extracting features, fine-tuning models, and conducting small-scale training. Also, the Raspberry Pi is particularly suitable in edge computing scenarios, where models trained on powerful systems are deployed for inference. Chapter 14 presents several AI model deployments and their applications on the Raspberry Pi 5, and Chapter 15 demonstrates how a custom-trained deep learning model can be deployed on a Raspberry Pi 5 for inference.

13.8 Exercise Problem

(Q1) Obtain a Raspberry Pi from your instructor and configure it for deep learning and AI applications by following the procedures outlined in this chapter. Once installed correctly, execute all the example problems provided in Chapter 14.

References

1 Gibbs, S. (2015). *Raspberry Pi Becomes Best Selling British Computer*. https://www.theguardian.com/technology/2015/feb/18/raspberry-pi-becomes-best-selling-british-computer.

2 Upton, E. (2023). *Raspberry Pi 5: Available Now*! https://www.raspberrypi.com/news/raspberry-pi-5-available-now/.

3 Raju, L., Sowmya, G., Srividhya, S. et al. (2021). Advanced home automation using Raspberry Pi and machine learning. In: *Proceeding 2021 7th International Conference on Electrical Energy Systems (ICEES)*, 600–605.

4 Manzi, F., Tuyishime, E., Hitayezu, A. et al. (2023). Leveraging machine learning and Raspberry Pi for enhanced wildlife remote monitoring and localization. In: *Proceeding 2023 International Conference on Modeling & E-Information Research, Artificial Learning and Digital Applications (ICMERALDA)*, 301–306.

5 Shalini, S., Arthi, R., Preetha, R. et al. (2021). Vehicle ignition control using Raspberry Pi and machine learning. In: *Proceeding 2021 International Conference on System, Computation, Automation and Networking (ICSCAN)*, 1–6.

6 Foundation, R.P. (2024). *Raspberry Pi 5 – Specifications*. https://www.raspberrypi.com/products/raspberry-pi-5/.

7 Debian (2024). *Debian "Bookworm" Release Information*. https://www.debian.org/releases/bookworm/.

8 Paszke, A., Gross, S., Chintala, S. et al. (2017). Automatic differentiation in PyTorch., NIPS 2017 Workshop on Autodiff, CA, USA. https://www.bibsonomy.org/bibtex/21530dd0202e55d3eb1ada151e09c499.

9 Bradski, G. (2000). The Opencv Library. *Dr. Dobb's Journal of Software Tools*, 120: 122–125.

10 Jeffrey, A.C. and Contributors (2024). *Pillow (PIL Fork) Documentation*. https://pillow.readthedocs.io/en/stable/.

11 ONNX (2024). *Open Neural Network Exchange*. https://onnx.ai/.

12 King, D.E. (2009). Dlib-ml: a machine learning Toolkit. *Journal of Machine Learning Research* 10: 1755–1758.

13 Geitgey, A. (2020). *Face-recognition 1.3.0*. https://pypi.org/project/face-recognition/.

14 PyPI (2024). *Pytesseract* 0.3.10. https://pypi.org/project/pytesseract/.

14

Deploy Deep Learning Models on Raspberry Pi

14.1 Introduction

This chapter explores various methods for deploying artificial intelligence (AI) models on a Raspberry Pi 5, which is a low-cost, compact device ideal for edge computing. To effectively implement the programs, it is essential to prepare the Raspberry Pi by following the setup instructions presented in Chapter 13. While the Raspberry Pi does not have the computational capacity to train deep learning models from the ground up, it is capable of running a variety of pre-trained models. By deploying these pre-trained models on the Raspberry Pi, we can utilize its capabilities to execute inference tasks at the edge, facilitating real-time AI applications without the need for powerful workstations or continuous cloud connectivity. To write Python codes on the Raspberry Pi, we can use the Thonny IDE, which can be launched from the menu icon. On the first launch, we can switch to "regular mode" by selecting the option in the top right corner. This process is shown in Figure 14.1 and requires Thonny to restart to apply the new setting.

14.2 Face Detection and Recognition in Video Feeds

14.2.1 Face Detection Using HOG

In this section, we will demonstrate a simple face-detection method in Raspberry Pi 5 using histogram of oriented gradients (HOG) [1]. This method extracts image features from an image by computing gradients and generating histograms of oriented gradients within small regions of the image. It can capture the shape and texture details of an image in a structured way. The extracted features are then used to train a support vector machine (SVM) model, which can classify regions of the image as faces or non-faces.

First of all, to implement real-time face detection using HOG, we have to import "OpenCV" and "dlib" libraries into Python. After that, we have to initialize the video capture object to access the USB camera, and the argument "0" in "cv2.VideoCapture(0)" refers to the default camera. We also need a HOG face detector object using "dlib"s "get_frontal_face_detector method", which is trained to detect frontal faces. In a continuous "while" loop, the code captures frames from the video stream, converts each frame from BGR to RGB color space to ensure compatibility with the HOG detector, and then applies the HOG face detector to the RGB frame to identify faces. This loop enables real-time face detection by continuously processing and detecting faces in the video feed (Figure 14.2).

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.



Figure 14.1 Launch Thonny IDE to write Python programs and switch to "regular mode."

```
example1.py
#import libraries
import cv2
import dlib

# Initialize the usb camera
video_capture = cv2.VideoCapture(0)

# Create a HOG face detector
hog_face_detector = dlib.get_frontal_face_detector()

while True:
    # Capture a frame from the video
    ret, frame = video_capture.read()

    # Convert the image from BGR color (OpenCV uses) to RGB color (dlib uses)
    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # Run hog face face detector on the frame
    faces = hog_face_detector(rgb_frame, 1)
```

Figure 14.2 Import libraries, initialize the USB camera, and enable real-time continuous face detection using a "while" loop.

```
    # Loop through each face we found in the frame
    for face_rect in faces:
        # Draw a box around the face
        cv2.rectangle(frame, (face_rect.left(), face_rect.top()), (face_rect.right(), face_rect.bottom()), (0, 0, 255), 4)

    # Display the processed frame with rectangles
    cv2.imshow('MyVideo', frame)

    # Hit 'q' on the keyboard to quit!
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release handle and destroy windows
video_capture.release()
cv2.destroyAllWindows()
```

Figure 14.3 Draw rectangles around every detected face, show the video frames, and break out from the while loop to close all windows opened by OpenCV.

Next, we draw rectangles around detected faces in a video frame and display the processed frame. It iterates over each detected face in the "faces" list, and code for the green color (0,0,255) is used to make green rectangles around every face using the "face_rect." The "cv2.rectangle" function draws the rectangle on the frame, and "cv2.imshow" displays the processed frame with these rectangles. The "while" loop waits for a keyboard input "q," and if pressed, it breaks the loop. Finally, the video capture object is released, and OpenCV closes all of its windows using "cv2.destroyAllWindows()." This last step is required to save system resources and exit the application properly. Figure 14.3 shows these codes in Python.

To verify the face detection process using HOG, open the Raspberry Pi terminal and execute the code from the directory containing the "example1.py" file. For example, as shown in Figure 14.4, the "example1.py" file was located in the "Desktop/examples" folder, so we navigated to that directory and ran the code from the "mydl" virtual environment, which has all the required packages. If you get an import library error, revisit the setup instructions in Chapter 13 to ensure all necessary packages are correctly installed.

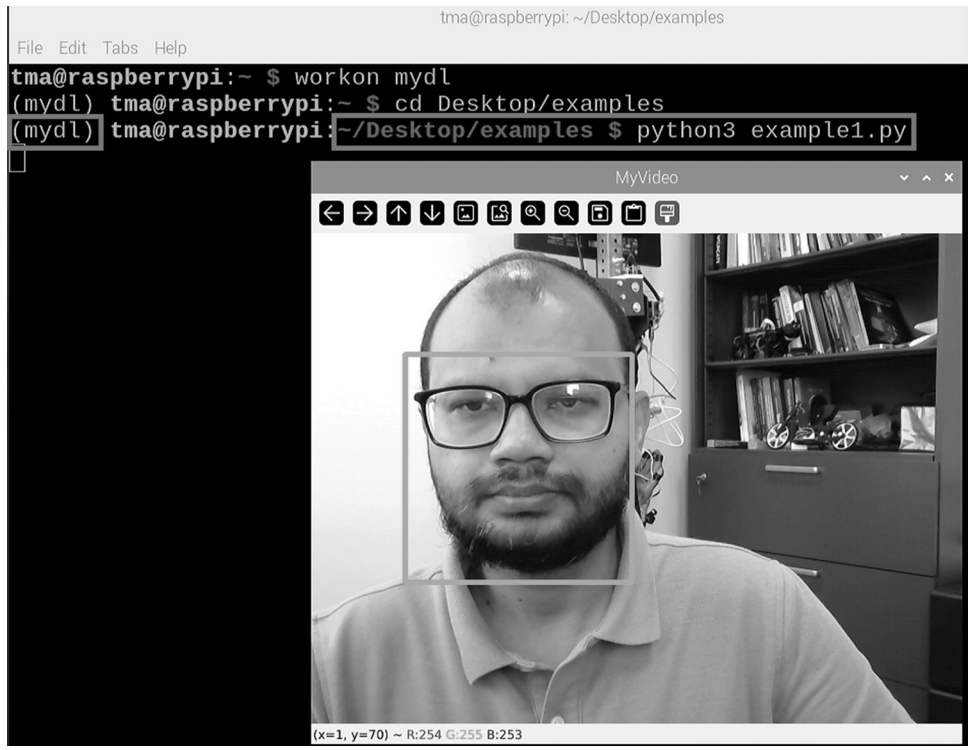


Figure 14.4 Run the "example1.py" python file from the "mydl" virtual environment to detect faces.

14.2.2 Face Recognition Using Transfer Learning

In this section, we will demonstrate a real-time face recognition process on a Raspberry Pi using transfer learning. We employed the Python library "face-recognition," which utilizes "dlib"'s deep learning models for face detection and recognition [2, 3]. "face-recognition" tool is popular among developers for its ease of implementation in facial recognition tasks, and since its first release in 2017, it has gone through several updates and bug fixes to improve accuracy, speed, and usability. "dlib" uses a deep convolutional neural network (CNN) for facial landmark detection and a deep metric learning model for face recognition. Also, these deep learning models in "dlib" have been trained on large datasets to accurately identify and verify facial features.

For face recognition, we will need some known faces so that the program can train on these to extract features and recognize these faces. We used three pictures (Messi and Ronaldo's photos are downloaded from the internet) for training and saved those inside the "Desktop/examples/face_recognition_images" directory (Figure 14.5). After that, we have to initialize the video capture object to access the USB camera, and the argument "0" in "cv2.VideoCapture(0)" refers to the default camera.

First, import the "face_recognition" and "cv2" (OpenCV) libraries to facilitate recognition and image processing tasks. Load these images from the specified directory using the "load_image_file" function of the "face_recognition" library. The "face_encodings" function is then applied to these images to compute a unique facial encoding (a numerical representation) of the facial features. These encodings are stored in the known_face_encodings 1, 2, and 3 variables, and the [0] indexing ensures that the first (and typically only) face encoding from the image is selected. These encodings can be utilized to identify the face in different images by matching it with

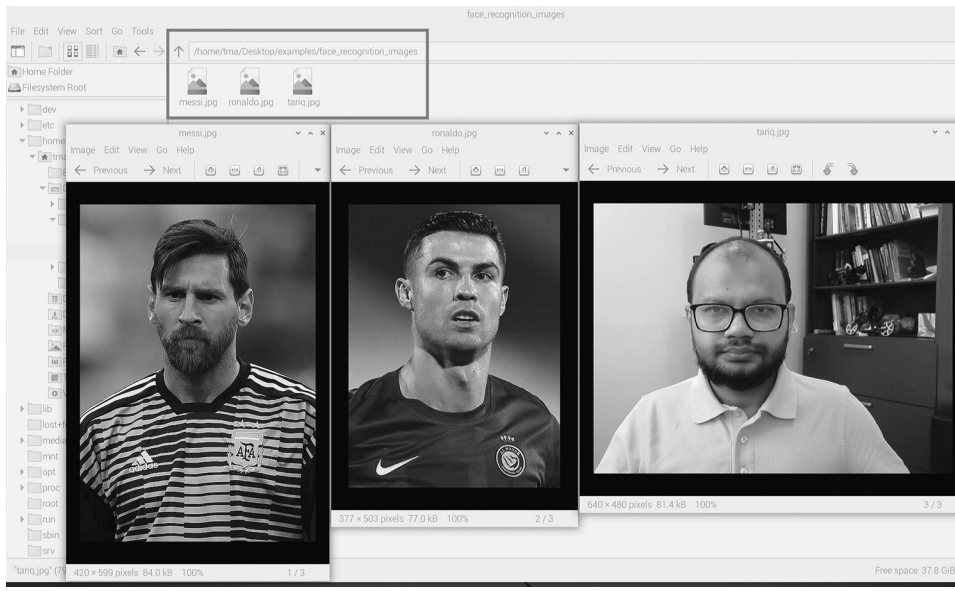


Figure 14.5 Save multiple known pictures in the "face_recognition_images" folder. Кирилл Венедиктов/ Wikimedia Commons / CC BY SA-3.0; Mehrdad Esfahani/SNN/Wikimedia Commons / CC BY 4.0.

```
#import libraries

import face_recognition
import cv2

# Load a images and learn how to recognize it using encodings
known_image1 = face_recognition.load_image_file("face_recognition_images/tariq.jpg")
known_face_encoding1 = face_recognition.face_encodings(known_image1)[0]

known_image2 = face_recognition.load_image_file("face_recognition_images/messi.jpg")
known_face_encoding2 = face_recognition.face_encodings(known_image2)[0]

known_image3 = face_recognition.load_image_file("face_recognition_images/ronaldo.jpg")
known_face_encoding3 = face_recognition.face_encodings(known_image3)[0]

# Initialize the webcam video
video_capture = cv2.VideoCapture(0)

# Create arrays of known face encodings and their names
known_face_encodings = [known_face_encoding1, known_face_encoding2, known_face_encoding3]
known_face_names = ["Tariq","Messi","Ronaldo"]
```

Figure 14.6 Load and encode known images and organize names in an array.

derived encodings (Figure 14.6). The "known_face_names" array is initialized with the names of the individuals whose faces correspond to the encodings: "Tariq," "Messi," and "Ronaldo." By organizing the face encodings and names in these arrays, the code sets up a reference system that allows the program to recognize and identify facial images.

A "while" loop is used to continuously capture frames from the video using the "read()" method. Since the captured frame by OpenCV uses BGR color format, it is converted to RGB format using "cv2.COLOR_BGR2RGB." The "face_recognition.face_locations" function is then used to detect the locations of all faces within the RGB frame. The "face_recognition.face_encodings" generates the facial encodings for each detected face. The code then iterates over each detected face using a for loop. For each face encoding, the "face_recognition.compare_faces" function checks if it matches any of the known face encodings stored in "known_face_encodings." If a match is found, the face is identified; otherwise, it is labeled as "Unknown." The name variable is initialized to "Unknown" and can be updated within the loop if a match is detected. This process is presented in Figure 14.7.

```
while True:
    # Capture a single frame
    ret, frame = video_capture.read()

    # Convert the image from BGR color (OpenCV) to RGB color (face_recognition uses)q
    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # Find all the faces and face encodings in the frame
    face_locations = face_recognition.face_locations(rgb_frame)
    face_encodings = face_recognition.face_encodings(rgb_frame, face_locations)

    # Loop through each face in this frame of video
    for (top, right, bottom, left), face_encoding in zip(face_locations, face_encodings):
        # See if the face is a match for the known face(s)
        matches = face_recognition.compare_faces(known_face_encodings, face_encoding)

        name = "Unknown"

        # If a match was found in known_face_encodings, use the first one.
        if True in matches:
            first_match_index = matches.index(True)
            name = known_face_names[first_match_index]
```

Figure 14.7 "while" loop to continuously capture frames, determine face locations, generate encodings, and match those encodings with known face encodings.

```
        # If a match was found in known_face_encodings, use the first one.
        if True in matches:
            first_match_index = matches.index(True)
            name = known_face_names[first_match_index]

        # Draw a box around the face
        cv2.rectangle(frame, (left, top), (right, bottom), (0, 0, 255), 2)

        # Draw a label with a name below the face
        cv2.rectangle(frame, (left, bottom - 35), (right, bottom), (0, 0, 255), cv2.FILLED)
        font = cv2.FONT_HERSHEY_DUPLEX
        cv2.putText(frame, name, (left+10, bottom-10), font, 1.0, (255, 255, 255), 1)

    # Display the resulting image
    cv2.imshow('Face Recognition', frame)

    # Hit 'q' on the keyboard to quit
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
# Release handle and destroy windows
video_capture.release()
cv2.destroyAllWindows()
```

Figure 14.8 Compare face encodings and create rectangles around known and unknown faces.

Figure 14.8 shows that if a match between frame face encodings and known face encodings is found, the program will create a red rectangle around the face using the "cv2.rectangle" function and put text on it using the "cv2.putText" function that displays corresponding known face names. If no matches are found, the default text on the frame will be "unknown."

Finally, the "cv2.imshow" displays the processed frames with rectangles showing known face names and "unknown" faces. The "while" loop waits for a keyboard input "q," and if pressed, it breaks the loop. Finally, the video capture object is released, and OpenCV closes all of its windows using "cv2.destroyAllWindows." Figure 14.8 shows these codes in Python.

Figure 14.9 shows that when this program (example2.py) is executed from the terminal, it continuously processes frames for face recognition. Using the Raspberry Pi USB camera, the program can accurately recognize a face, and when aimed at a YouTube video, it successfully identifies players whose names and images were used for training. It is important to note that some faces in Figure 14.9 are labeled as "unknown" as the model has never seen their images. However, the overall model accuracy is quite good.



Figure 14.9 Face recognition using video streams from USB camera and computer monitor.

A major advantage of the "face-recognition" library is its ability to efficiently automate the identification of individuals in photos or videos. This widely used library is popular for verifying identities for security purposes and developing interactive applications that respond to facial recognition.

14.3 Real-time Object Detection

14.3.1 Object Detection Using YoloV3

In this section, we will demonstrate a transfer learning approach for real-time object detection using the pre-trained model you only look once, version 3 (YOLOv3). YOLOv3 was developed by Joseph Redmon and Ali Farhadi in 2018, and it is a very popular optimized model [4]. The model is suitable for applications that require fast and accurate object detection from the common objects in context (COCO) dataset, which contains 80 different objects ranging from common everyday items to animals and vehicles.

To implement the YOLOv3 model, we downloaded the model configuration file (yolov3.cfg), the weights (yolov3.weights), and the label names from the COCO dataset (coco.names). The download links for these resources are provided in Table 14.1.

These files are stored in the same directory as the Python file (example3.py). In our case, this is the "Desktop/example" directory on the Raspberry Pi. In the "example3.py" program, we first import the "OpenCV" and "numpy" libraries, and the YOLOv3 model for object detection was loaded using OpenCV's deep neural network (dnn) module (cv2.dnn.readNet()). The program then reads the class labels from the "coco.names" file and stores each label in the "classes" list. The "getLayerNames()" method retrieves all the layer names of the neural network, which are stored in "layer_names." For each index, it appends the corresponding layer name to the "output_layers" list and captures video from the USB camera (Figure 14.10).

Table 14.1 Download the YOLOv3 model's configuration, weights, and COCO label names.

Item	Download location
yolov3.cfg	https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg
yolov3.weights	https://github.com/patrick013/Object-Detection---Yolov3/blob/master/model/yolov3.weights
coco.names	https://github.com/pjreddie/darknet/blob/master/data/coco.names

example3.py

```
#import libraries
import cv2
import numpy as np

# Load YOLO model and weights
net = cv2.dnn.readNet( "yolov3.cfg","yolov3.weights")
with open("coco.names", "r") as f:
    classes = [line.strip() for line in f.readlines()]

layer_names = net.getLayerNames()
# define empty output layers
output_layers = []
for i in net.getUnconnectedOutLayers():
    index = i-1
    # append output layers
    output_layers.append(layer_names[index])
# Initialize video capture
video_capture = cv2.VideoCapture(0)  # 0 is the default usb camera
```

Figure 14.10 Load the YOLOv3 model configuration, weights, and labels, and read the model to append corresponding layer names to the "output_layers" list.

```
while True:
    # Capture frame-by-frame and initialize height, width, and channels
    ret, frame = video_capture.read()
    height, width, channels = frame.shape
    # Detecting objects using loaded network
    my_blob = cv2.dnn.blobFromImage(frame, 1/255, (416, 416), (0, 0, 0), True, crop=False)
    net.setInput(my_blob)
    outs = net.forward(output_layers)

    # Information to show on the screen
    class_ids = []
    confidences = []
    boxes = []
    # Loop through output frames
    for output in outs:
        for detection in output:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class id]
```

Figure 14.11 The real-time object detection using the YOLOv3 model is performed in a continuous "while" loop.

```
            if confidence > 0.6:
                # Object detected
                center_x = int(detection[0] * width)
                center_y = int(detection[1] * height)
                w = int(detection[2] * width)
                h = int(detection[3] * height)
                # Create rectangle coordinates
                x = int(center_x - w / 2)
                y = int(center_y - h / 2)
                boxes.append([x, y, w, h])

                confidences.append(float(confidence))
                class_ids.append(class_id)

    # remove overlapping boxes
    indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
```

Figure 14.12 Implement the logic for creating bounding boxes around detected items using the confidence level.

The real-time object detection using the YOLOv3 model is performed in a continuous "while" loop (Figure 14.11). A blob using "cv2.dnn.blobFromImage", which scales the pixel values, resizes the frame to 416×416 and normalizes it. The blob is then set as the input to the network with "net.setInput(my_blob)." The "net.forward(output_layers)" function performs the forward pass through the network, producing output layers with detection information.

From the network output, the class with the highest score is identified using "np.argmax(scores)," and its confidence is stored in the "confidence" variable. If the confidence level is above 0.6, the object is considered detected. For each box that is in the index list, the bounding box coordinates are retrieved. These steps in the Python program are shown in Figure 14.12.

The bounding box center coordinates (center_x, center_y) and dimensions (w, h) are calculated relative to the frame's width and height and drawn on top of the detected items using "cv2.rectangle." "cv2.putText" is used to put the classified label names and confidence level on top of the bounding boxes.

The "cv2.imshow" displays the processed frames with rectangles showing known objects. The "while" loop waits for a keyboard input "q," and if pressed, it breaks the loop. Finally, the video capture object is released, and OpenCV closes all of its windows using "cv2.destroyAllWindows." Figure 14.13 shows these codes in Python.

Figure 14.14 shows real-time inference using the YOLOv3 model on a Raspberry Pi. We will need to run this Python code (example3.py) within the virtual environment and from the directory containing the Python programming files, model configurations, weights, and label names. If these files are in a different directory, we will need to specify the correct folder location in the

```
    # remove overlapping boxes
    indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
    # Draw bounding boxes
    for i in range(len(boxes)):
        if i in indexes:
            x, y, w, h = boxes[i]
            label = str(classes[class_ids[i]])
            confidence = confidences[i]*100
            cv2.rectangle(frame, (x, y), (x + w, y + h), (255,0,255), 2)
            cv2.putText(frame, f"{label} {confidence:.1f}%", (x, y - 5), cv2.FONT_HERSHEY_SIMPLEX, 1, (255,0,255), 2)

    # Display the resulting frame
    cv2.imshow('Frame', frame)
    # Exit if 'q' is pressed on the video frame
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
# Release the capture and close windows
video_capture.release()
cv2.destroyAllWindows()
```

Figure 14.13 Create bounding boxes around detected objects and display confidence level.

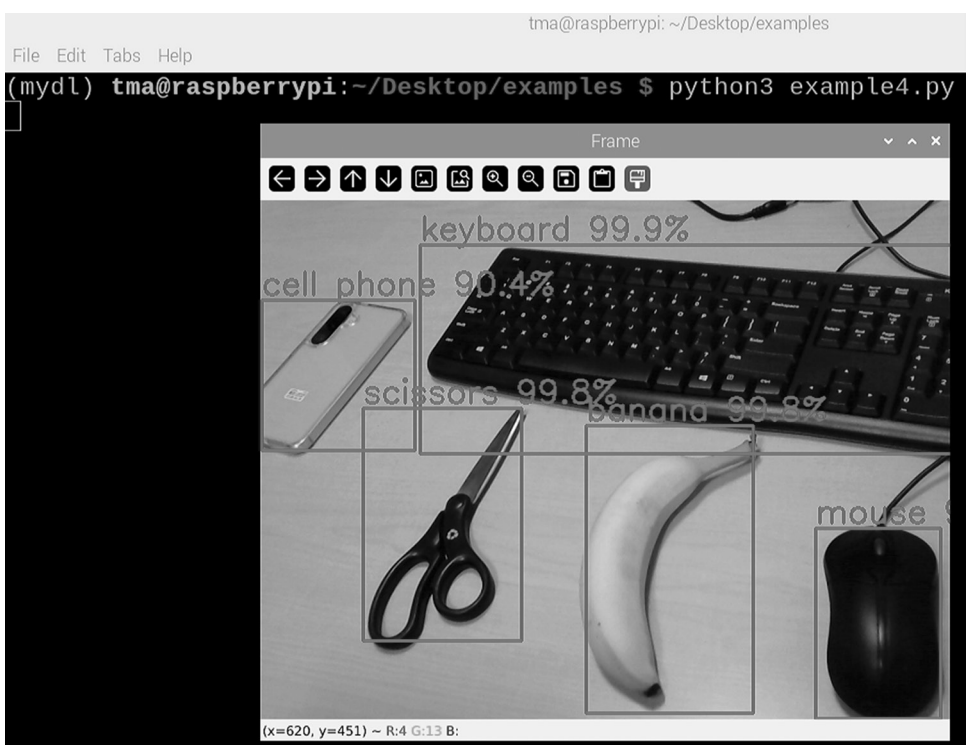


Figure 14.14 Object detection using the YOLOv3 and corresponding confidence level.

Python program. The program effectively detects objects from a USB video stream. However, the model is trained to recognize only 80 different objects. Attempting to detect objects outside of these 80 categories will not work on this pre-built network. We may use a customized model to detect objects other than COCO items in Raspberry Pi. In that scenario, we have to create a dataset and train the model separately on a desktop computer as shown in Chapter 8–10, since Raspberry Pi's limited computing resource are not suitable for most deep learning training.

The YOLOv3 used in this section is a well-trained and optimized model. Due to extensive training on this model, it is highly effective in real-world scenarios having those 80 class objects. We can see from the output frame that it can provide robust object detection capabilities in real-time camera feed. Also, this model can be integrated into various applications of embedded devices, such as surveillance systems, autonomous vehicles, and image analysis.

14.4 Real-time Classification

14.4.1 Classification Using PyTorch's Quantized Model

In this section, we will show how PyTorch's quantized model of a deep learning network can be utilized in Raspberry Pi for real-time classification applications. Many popular deep learning models have quantized versions developed by PyTorch, where the weights and activations are represented with lower precision, typically using 8-bit integers instead of the standard 32-bit floating-point numbers. Although quantized models have reduced precision levels, they are suitable for low-powered embedded devices, such as Raspberry Pi. These models can perform real-time inference more efficiently when processing power is limited. They can also minimize latency during inference on embedded devices while maintaining a satisfactory level of accuracy.

For the classification task, we have used a pre-trained deep learning network "mobilenet_v2", and the classification results are displayed in the video output. First, we import "PyTorch," "torchvision," "OpenCV," "NumPy," and "PIL" libraries and set the quantized engine to "quantized neural network package (QNNPACK)" for optimizing inference on embedded devices. The video capture from the USB camera was initialized and also the frame width, height, and frames per second (FPS) were set to specific values. For the image transformation, images need to be converted to tensors and normalized. This was done by defining a callable object "preprocess." The pre-trained and quantized "mobilenet_v2" is loaded from "torchvision" using a variable called "net." These steps are presented in Figure 14.15.

example4.py

```
#import libraries
import torch
from torchvision import models, transforms
import cv2
import numpy as np
from PIL import Image

# set the quantized engine to QNNPACK
torch.backends.quantized.engine = 'qnnpack'

# capture video and set frame width, height, and fps
video_capture = cv2.VideoCapture(0, cv2.CAP_V4L2)
video_capture.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
video_capture.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
video_capture.set(cv2.CAP_PROP_FPS, 36)

#combine several image transformation operations into a callable object
preprocess = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

#internet connection is required to download  the model
net = models.quantization.mobilenet_v2(pretrained=True, quantize=True)
```

Figure 14.15 Import libraries, set quantized engine to QNNPACK, capture video, create "preprocess" object for image transformation, and load the pre-trained model.

Next, the model is converted to the TorchScript model to reduce Python dependencies, and a function is defined to load class names from the "imagenet" text file, which can be stored in the variable "classes." A "while" loop continuously processes video frames captured from the webcam. For each frame, it reads the frame, converts the image from BGR to RGB, preprocesses the image using the defined transformations, and creates a mini-batch for model input (Figure 14.16).

Figure 14.17 shows that to process the output from the model, the softmax function is applied to convert raw output scores into probabilities. "top.sort" is used to sort the list of tuples in descending order based on the probability. The "key=lambda" specifies that the sorting should be done based on the second element (probability).

The "cv2.putText" function is utilized to overlay the top classified label name and confidence level onto the output frame. The "cv2.imshow" function displays these processed frames continuously while the while loop is active. The program waits for a keyboard input "q," and upon detection, it terminates the "while" loop. Finally, the video capture object is released, and OpenCV closes all its windows using "cv2.destroyAllWindows." Figure 14.17 illustrates these codes in Python and Figure 14.18 shows the output frames after inference.

We observe that this pre-trained model performs effectively and can provide real-time inference on the camera feed. It's important to note that not all PyTorch models are pre-trained with "qnnpack" and suitable for the Raspberry Pi. However, we can convert pre-trained models into a quantized form and make it highly efficient for embedded device applications.

```
# PyTorch to a TorchScript model
net = torch.jit.script(net)

#define function to read classes from text file
def load_class_names(file_path):
    with open(file_path,'r') as file:
        class_names = [line.strip() for line in file.readlines()]
    return class_names
classes = load_class_names('imagenet1000_clsidx_to_labels.txt')

with torch.no_grad():
    while True:
        # read frame
        ret, image = video_capture.read()
        # convert opencv output from BGR to RGB
        rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        # preprocess RGB image
        input_tensor = preprocess(rgb_image)
        # create a mini-batch
        input_batch = input_tensor.unsqueeze(0)
        # run input batch through the model
        output = net(input_batch)
```

Figure 14.16 Convert the model to TorchScript, load class names from the "imagenet100_clsidx_labels.txt" text file, use the "while" loop to continuously process video frames, and create a mini-batch for model input.

```
        # get top prediction and sort it
        top = list(enumerate(output[0].softmax(dim=0)))
        top.sort(key=lambda x: x[1], reverse=True)

        # draw the top prediction on the frame
        for idx, (class_idx,score) in enumerate(top[:1]):
            text_obj = f"{classes[class_idx][6:]}".rstrip("',")
            text_conf = f"Confidence: {score.item()*100:.1f}%"
            x,y= 20,30 # text position
            cv2.putText(image, text_obj, (x,y), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 255), 2)
            cv2.putText(image, text_conf, (x,y+35), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 255), 2)
        # display the frame
        cv2.imshow('Video', image)
        # Exit if 'q' is pressed on the video frame
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
# Release video capture and close windows
video_capture.release()
cv2.destroyAllWindows()
```

Figure 14.17 Process the model output in the "while" loop to sort the predicted list of tuples in descending order, so the highest probabilities come first.

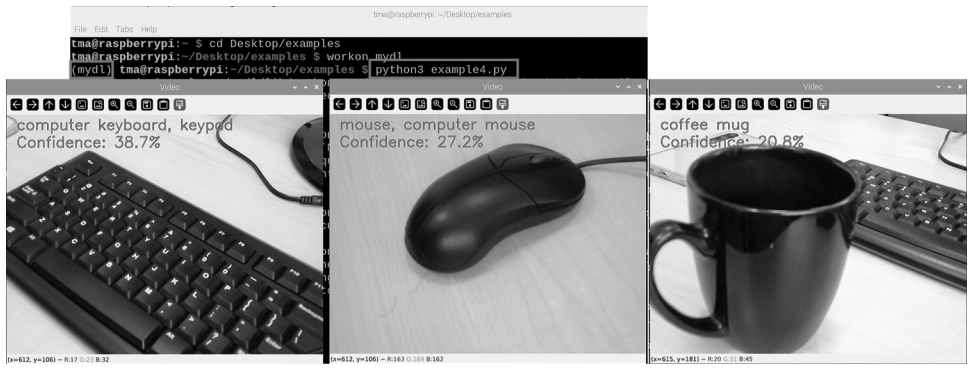


Figure 14.18 Classification of video frame objects using the "mobilenet_v2" model.

14.5 Real-time Segmentation

14.5.1 Segmentation Using *k*-means Clustering

k-means clustering is a popular unsupervised machine learning model that can partition a dataset into *k* distinct clusters. It works by assigning each data point to the group with the closest average value. The algorithm starts with random group centers and improves them through iteration, and this process repeats until convergence [5]. When applied to image segmentation on compact devices, such as the Raspberry Pi, *k*-means can group pixels by their similarity of colors or how close they are to each other. By sorting pixels into *k* segments, we can effectively outline different areas in an image. This technique is computationally efficient and can be implemented on resource-constrained devices, such as the Raspberry Pi. The segmentation task can be very useful in identifying objects, removing backgrounds, or simplifying images or videos for further analysis.

To implement the real-time segmentation in the video feed from the camera, we first import "OpenCV," "dlib," "numpy," and "time" libraries, and capture video from the USB camera for initialization. The video frame was resized to speed up the calculation and converted to BGR to RGB format so that the "dlib" library can use the frames. The criteria for *k*-means clustering are defined, which include the type of termination criteria, the maximum number of iterations (100), and the desired accuracy (0.2) (Figure 14.19).

Figure 14.20 shows that the "cv2.kmeans" function performs *k*-means clustering on the flattened image with 4 clusters (k=4). This function returns the labels for each pixel and the cluster centers, and a set of unique colors (red, green, blue, and yellow) is defined to represent each cluster. The segmented image is created by mapping each pixel's cluster label to its corresponding color (Figure 14.20).

After the *k*-means clustering, the segmented image is reshaped to the original frame's shape and converted back to BGR color space for OpenCV display. For the final output, the original and segmented images are overplayed using "cv2.addWeighted," with weights of 0.6 for the original image and 0.4 for the segmented image. The "cv2.imshow" function displays original, segmented, and overplayed frames continuously while the "while" loop is active. The "time.sleep" is used to pause for one second inside the "while" loop. The program waits for a keyboard input "q," and upon detection, it terminates the "while" loop. Finally, the video capture object is released, and OpenCV closes all its windows using "cv2.destroyAllWindows." Figure 14.21 illustrates these codes in Python.

example5.py

```python
#import libraries
import cv2
import dlib
import numpy as np
import time

# Initialize the usb camera
video_capture = cv2.VideoCapture(0)

while True:
    # Capture a frame from the video
    ret, frame = video_capture.read()
    #resize the image for faster calculation (optional)
    frame = cv2.resize(frame, (400, 400))
    # Convert the image from BGR color (OpenCV uses) to RGB color (dlib uses)
    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    # Flatten the image
    flat_frame = np.float32(rgb_frame.reshape(-1, 3))
    # Define criteria for k-means clustering
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)
```

Figure 14.19 Import libraries, initialize video stream, and use a "while" loop to continuously process video frames for *k*-means clustering.

```python
    k = 4  # Choose the Number of clusters
    _, labels, centers = cv2.kmeans(flat_frame, k, None, criteria, 8, cv2.KMEANS_RANDOM_CENTERS)
    # Define a set of unique colors for each cluster
    colors = [
        [255, 0, 0],   # Red
        [0, 255, 0],   # Green
        [0, 0, 255],   # Blue
        [255, 255, 0]  # Cyan
    ]
    # Map each cluster to its corresponding color
    colored_centers = np.array(colors, dtype=np.uint8)
    segmented_image = colored_centers[labels.flatten()]
    segmented_image = segmented_image.reshape(rgb_frame.shape)

    # Convert segmented image back to BGR for OpenCV display
    segmented_image = cv2.cvtColor(segmented_image, cv2.COLOR_RGB2BGR)
```

Figure 14.20 Segmentation using *k*-means clustering and mapping segmented labels to different colors.

```python
    # Convert segmented image back to BGR for OpenCV display
    segmented_image = cv2.cvtColor(segmented_image, cv2.COLOR_RGB2BGR)

    # Blend the original and segmented images
    alpha = 0.6  # Weight of the original image
    beta = 0.4   # Weight of the segmented image
    overlay_image = cv2.addWeighted(frame, alpha, segmented_image, beta, 0)

    # Display the processed frame with rectangles
    # Display results
    cv2.imshow('Original', frame)
    cv2.imshow('Segmented', segmented_image)
    cv2.imshow('Overlay', overlay_image)
    # have some rest
    time.sleep(1)
    # Hit 'q' on the keyboard (whlie video window is active) to quit!
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release handle and destroy windows
video_capture.release()
cv2.destroyAllWindows()
```

Figure 14.21 Convert segmented frame from RGB to BGR and display original, segmented, and overplayed frames continuously using the "while" loop.

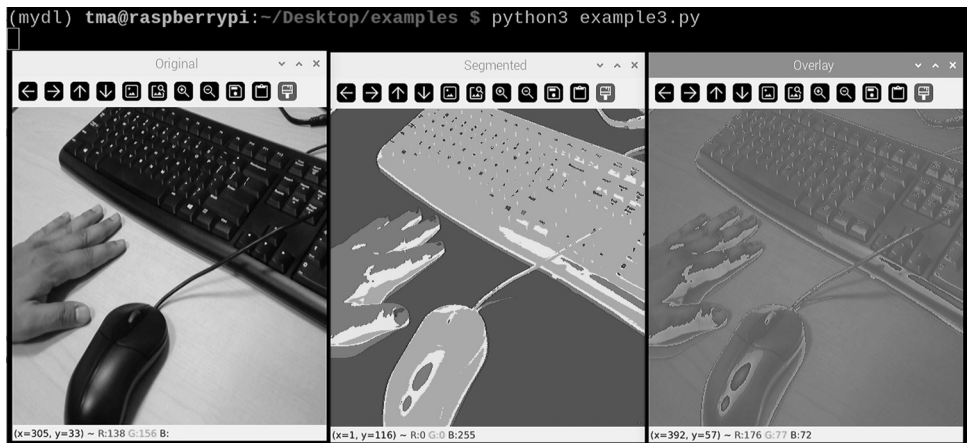


Figure 14.22 Original, segmented, and overlay frames from the video stream during the segmentation task using *k*-means clustering.

Figure 14.22 shows the original, segmented, and overlay frames from the video stream. We observe that the *k*-means clustering segmentation process on the Raspberry Pi is quite fast. Different colors are utilized in the segmented regions to enhance the visualization of the results.

14.6 Exercise Problems

(Q1) Section 14.2.1 demonstrates how to use a HOG model for face detection. It utilizes the "dlib" library, which includes machine learning algorithms and tools for solving computer vision problems. The ".dat" models from "dlib" can be downloaded from their GitHub page at https://github.com/davisking/dlib-models/. In this exercise, you will develop a face landmark detection program to identify landmarks in a video stream using one of "dlib"'s shape prediction models. If implemented correctly, the output video from your program should be similar to Figure 14.23.

(Q2) Section 14.4.1 utilizes PyTorch's "mobilenet_v2" model for classifying video frames. In this example, replace it with the "mobilenet_v3_large" quantized model for a similar task. Determine which of these two models performs better in terms of accuracy in identifying similar objects. Your program's output should look like Figure 14.24 (here, the confidence level is hidden).

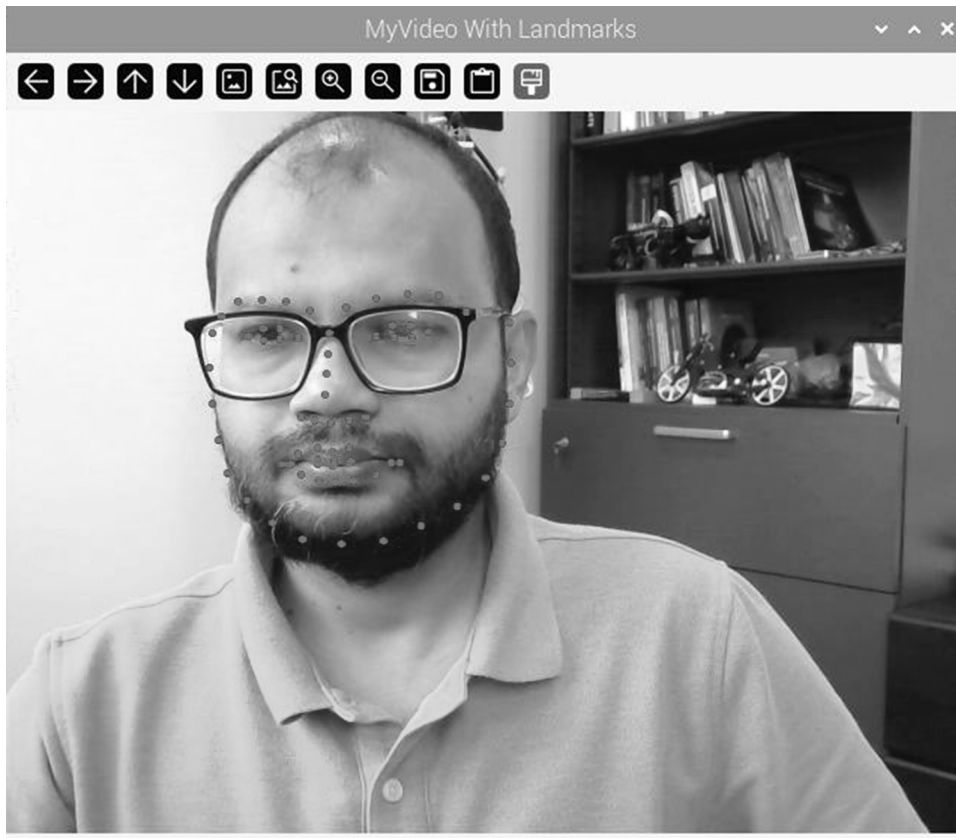


Figure 14.23 Face landmark detection video using "dlib"'s shape predictor model.

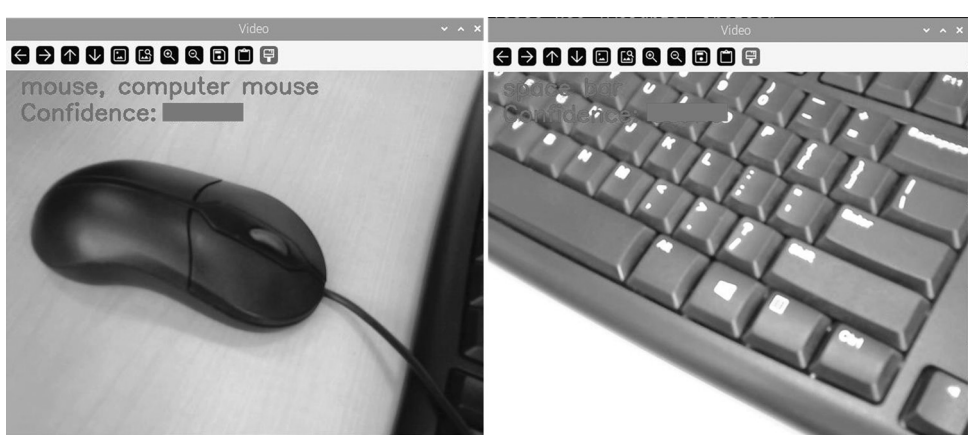


Figure 14.24 Classification of video frame objects using the "mobilenet_v3_large" model.

References

1 Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. *Proceeding IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 881: 886–893.

2 Geitgey, A. (2016). *Face Recognition*. https://face-recognition.readthedocs.io/en/latest/index.html.

3 King, D.E. (2009). Dlib-ml: a machine learning toolkit. *Journal of Machine Learning Research* 10: 1755–1758.

4 Redmon, J. and Farhadi, A. (2018). YOLOv3: an incremental improvement. *arXiv*.abs/1804.02767.

5 Steinley, D. and Brusco, M.J. (2007). Initializing *k*-means batch clustering: a critical evaluation of several techniques. *Journal of Classification* 24 (1): 99–121.

15

Trained PyTorch Model: From Desktop PC to Raspberry Pi

15.1 Introduction

Transferring a trained PyTorch model to compact and low-cost embedded devices such as Raspberry Pi is highly useful for edge computing and Internet of Things (IoT) applications. In many robotic platforms, drones, and portable devices, carrying a powerful computer for real-time inference is not feasible for flexible operations. Also, challenges such as connectivity issues, latency, or privacy concerns may arise when relying on cloud-based services. In these situations, transferring a PyTorch model (in .pt or .pth format) to the Raspberry Pi enables on-device inference without needing external hardware or cloud services.

The Raspberry Pi's low-cost, minimal power usage, portability, and wide availability make it an excellent choice for deploying artificial intelligence (AI) models. However, due to its limited processing power, PyTorch models must be optimized for proper execution. A common approach is to convert the PyTorch model to open neural network exchange (ONNX) format for deployment. This will provide a standardized representation that can be further compressed and accelerated using tools specifically tailored for Raspberry Pi. By enabling PyTorch model deployment on Raspberry Pi, developers can create intelligent, standalone systems that bring machine learning capabilities to various applications, from smart farming to industrial IoT solutions.

15.2 Model Training on a Desktop PC

In this chapter, we will demonstrate how to transfer an object detection model from a desktop PC to a Raspberry Pi 5. The process of training an object detection model using deep transfer learning is detailed in Chapter 10, where we utilized a dataset with four distinct classes (pencil, scissors, wrench, and screwdriver) for object detection and image classification tasks. The custom model developed in that chapter is saved in the "classification1" as "best_fasterrcnn_model.pth." For a detailed explanation of the training process, review Chapter 10.

15.3 PyTorch's.pth to ONNX

15.3.1 ONNX Runtime

Before we proceed with model conversion, we need to install the "onnxruntime" library on Raspberry Pi 5. This library should be installed within the "mydl" virtual environment, which

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.
© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

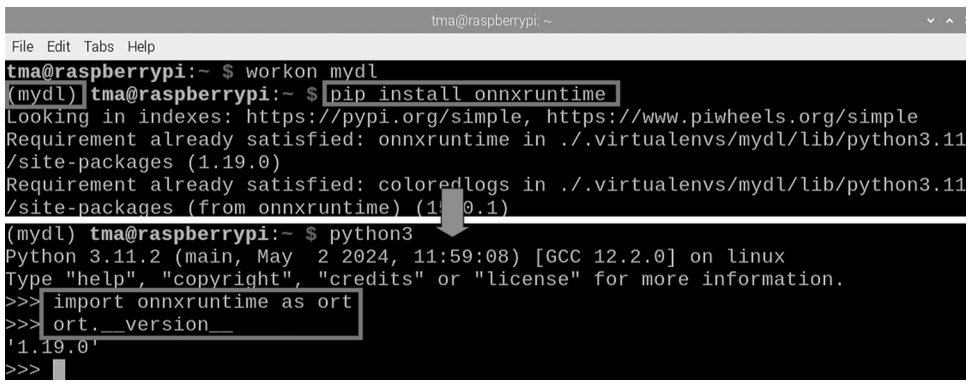


Figure 15.1 Install "onnxruntime" and check the version in Python.

will allow us to use other tools, such as OpenCV and Torch, that were set up in the Raspberry Pi setup (Chapter 13). "onnxruntime" allows models trained in various frameworks, such as PyTorch and TensorFlow, to run smoothly on Raspberry Pi by converting them to the ONNX format. It optimizes model inference by enabling hardware acceleration and model compression techniques, resulting in faster execution and reduced memory usage [1]. To install this package, execute "pip install onnxruntime" inside the virtual environment "mydl" as shown in Figure 15.1. Make sure to enter the "mydl" environment (created in Chapter 13) before this installation process. To verify the installation, enter "python3" to run the Python interpreter and import the "onnxruntime" library, and check the version (Figure 15.1).

15.3.2 Model Conversion in Raspberry Pi 5

For the model conversion task, we copied the trained PyTorch object detection model, "best_fasterrcnn_model.pth," from Chapter 10. This file is located on the Raspberry Pi's Desktop at the path "/home/tma/Desktop/pc2raspi." The Python script used for the conversion, named "pth2onnx.py," is also stored in the same folder, as shown in Figures 15.2 and 15.3.

First, we set up the pre-trained custom FasterRCNN with ResNest-101 backbone using PyTorch and "torchvision" libraries. The FasterRCNN model is imported from "models.detection.faster_rcnn" and the "resnet_fpn_backbone" is imported from the "models.detection.backbone_utils" modules of the "torchvision" library. The "torch" module loads the state dictionary from the trained model "best_fasterrcnn_model.pth" and adds the pre-trained weights to the ResNet101-FPN backbone configured for five classes. The model is then set to evaluation mode for inference. A dummy input tensor is created to represent an 800 × 800 RGB image, matching the model's expected input size as shown in Figure 15.2.

The "torch.onnx.export()" function is utilized to convert the PyTorch model into an ONNX model. In this process, the pre-trained model is transformed to the ONNX format based on the provided "dummy_input" and saved in the same directory (Model_PATH+"/best_fasterrcnn_model.onnx"). The "opset_version=11" argument ensures that the model is compatible with ONNX version 11, and the "input_names" and "output_names" specify the names of the input and output tensors. The "dynamic_axes" parameter allows the batch size to remain flexible for the input and output tensors (boxes, labels, and scores). After the conversion, the script displays

pth2onnx.py

```
#import libraries
import torch
import torchvision
from torchvision.models.detection.faster_rcnn import FasterRCNN
from torchvision.models.detection.backbone_utils import resnet_fpn_backbone

Model_PATH = r'/home/tma/Desktop/pc2raspi'
#Load state dictionary to cpu
state_dict = torch.load(Model_PATH+'/best_fasterrcnn_model.pth', map_location=torch.device('cpu'))

# Create FasterRCNN model with ResNet101-FPN backbone
backbone = resnet_fpn_backbone('resnet101', pretrained=False)
model = FasterRCNN(backbone, num_classes=5)  # Change this num_classes if necessary
                                             # Currently, 4 classes + 1 for background
# Load the state dictionary into the model
model.load_state_dict(state_dict)
# Set the model to evaluation mode
model.eval()

# Create a dummy input using image size
dummy_input = torch.randn(1, 3, 800, 800) #The .pth model expects 800 by 800 images

```

Figure 15.2 Import libraries, load state dictionary from the trained model, and setup FasterRCNN with ResNet101-FPN backbone.

```
# Export the model to ONNX
torch.onnx.export(model, dummy_input, Model_PATH+"/best_fasterrcnn_model.onnx",
                  opset_version=11,
                  input_names=['input'],
                  output_names=['boxes', 'labels', 'scores'],
                  dynamic_axes={'input': {0: 'batch_size'},
                                'boxes': {0: 'batch_size'},
                                'labels': {0: 'batch_size'},
                                'scores': {0: 'batch_size'}})

print(f"ONNX conversion is successful! Model saved at {Model_PATH}")
```

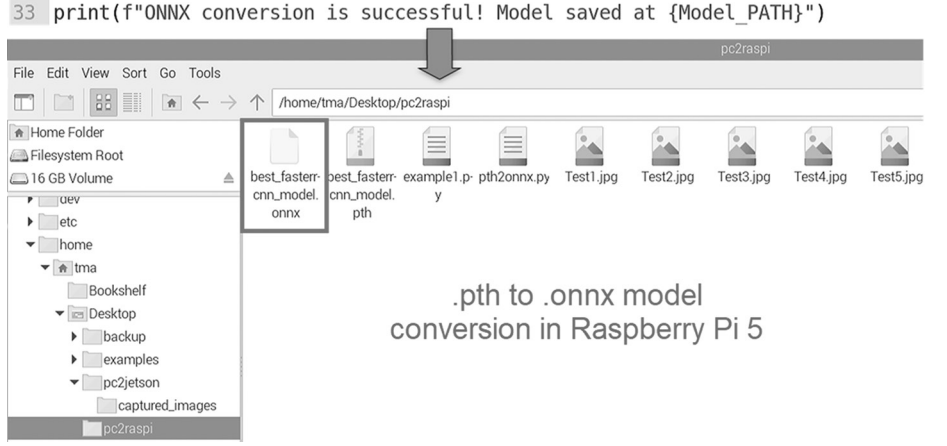


Figure 15.3 Convert the PyTorch model into an ONNX model and save it in a specified directory using the "torch.onnx.export()" function.

a confirmation message indicating that the model has been successfully saved in ONNX format. This conversion facilitates optimized inference on the Raspberry Pi. As shown in Figure 15.3, once the program runs, the ONNX model (best_fasterrcnn_model.onnx) is saved into the same directory.

15.4 ONNX Model Inference on Raspberry Pi 5

15.4.1 Import Libraries and Define Configurations

Once the ONNX model (model.onnx) is generated, the next step is to use this model on the Raspberry Pi for real-time inference. It is recommended to perform the ONNX model conversion directly on a Raspberry Pi 5 rather than a desktop PC, as "onnxruntime" might conflict with the existing version of PyTorch on a desktop machine. In this example, the newly created ONNX model is saved in the "pc2raspi" directory on the Raspberry Pi's Desktop. For the model inference, we created a new Python script, named "example1.py," in the same directory using the Thonny editor.

First, we import the necessary libraries, including OpenCV, NumPy, and "onnxruntime" to handle image processing and testing a ".jpg" test image. We configure the paths to both the pre-trained ONNX model (best_fasterrcnn_model.onnx) and the input image (Test1.jpg). It also defines the input image size as 800×800 pixels, which is the expected dimension for model input. The "Class_NAMES" dictionary maps numerical labels to specific object classes (pencil, scissors, screwdriver, and wrench), which will be used to identify detected objects in the image. We also define a "Confidence_THRESHOLD" variable, and it is set to 0.7, meaning the model will only consider object detections with a confidence score of 70% or higher (Figure 15.4). This will help filter out predictions by the model that are less certain.

15.4.2 Image Preprocessing

To preprocess a test image, the "preprocess_image" and "rescale_boxes" functions are defined. The "preprocess_image" function prepares an input image for the ONNX model by loading it, resizing it to a predefined input size, and converting it to RGB color space. Next, the pixel values are normalized and transported to the dimensions to match the expected input format. After image processing, the "preprocess_image" function returns the original image, the preprocessed image tensor, and the original image dimensions. The "rescale_boxes" function adjusts the bounding box coordinates predicted by the model back to the original image dimensions. It applies scaling factors to each coordinate to ensure the detected objects are correctly located as found in the original image. These functions are shown in Figure 15.5.

example1.py

```
1  # Import libraries
2  import cv2
3  import numpy as np
4  import onnxruntime
5  import sys
6
7  # Configuration
8  Model_PATH = "/home/tma/Desktop/pc2raspi/best_fasterrcnn_model.onnx"
9  Image_PATH = "/home/tma/Desktop/pc2raspi/Test1.jpg"
10 Input_SIZE = (800, 800)
11 Class_NAMES = {1: 'Pencil', 2: 'Scissors', 3: 'Screwdriver', 4: 'Wrench'}
12 Confidence_THRESHOLD = 0.7
```

Figure 15.4 Import libraries and configure model variables.

```
# Function for preprocessing the image
def preprocess_image(image_path):
    image = cv2.imread(image_path)
    if image is None:
        print(f"Failed to load image: {image_path}")
        sys.exit(1)

    original_size = image.shape[:2]
    image_resized = cv2.resize(image, Input_SIZE)
    image_rgb = cv2.cvtColor(image_resized, cv2.COLOR_BGR2RGB)
    image_normalized = image_rgb.astype(np.float32) / 255.0
    image_transposed = np.transpose(image_normalized, (2, 0, 1))
    return image, np.expand_dims(image_transposed, axis=0), original_size

# Function for rescaling bounding boxes
def rescale_boxes(boxes, original_size):
    y_scale = original_size[0] / Input_SIZE[0]
    x_scale = original_size[1] / Input_SIZE[1]
    scaled_boxes = boxes.copy()
    scaled_boxes[0::4] *= x_scale  # x_min
    scaled_boxes[1::4] *= y_scale  # y_min
    scaled_boxes[2::4] *= x_scale  # x_max
    scaled_boxes[3::4] *= y_scale  # y_max
    return scaled_boxes
```

Figure 15.5 "preprocess_image" and "rescale_boxes" functions to process images of varying sizes and then accurately map the detection results back onto the original images.

15.4.3 Model Inference with Bounding Boxes

We created a "draw_box" function for drawing bounding boxes around detected objects after the inference. It takes the image, the bounding coordinates, the class name of the detected object, and the confidence score as inputs. The function then converts the box coordinates to integers and generates a random color for each detected class. Finally, it draws a rectangle around the detected object using "cv2.rectangle." To overlay the class names and confidence scores as labels above the box, the "cv2.putText" function is used.

The Python program then loads the ONNX model using the "onnxruntime.InferenceSession" to perform inference. The image is preprocessed (resized and converted to a tensor) using the "preprocess_image" function, and the inference is run using the model session. After that, the outputs from the model (boxes, labels, and confidence scores) are extracted and processed. The bounding boxes from the model output are flattened and rescaled to the original image size using the "rescale_boxes" function. And those boxes are reshaped into a 2D array where each row corresponds to a box. All detections by the model are then filtered based on the confidence threshold, which keeps only those with a confidence score above the predefined "Confidence_THRESHOLD" value of 0.7. These programming steps are shown in Figure 15.6.

Next, we check whether there are any valid detections by evaluating the "valid_detections" array. If valid detections exist, it creates a copy of the original image (result_image) to draw the bounding boxes and print labels. For each valid detection, it retrieves the bounding box coordinates, label, and confidence scores and maps the label to a class name using the "Class_NAMES" dictionary. The "draw_box" function is then used to draw bounding boxes and put labels on the "result_image," and the detection details (class name, confidence score, and bounding box coordinates) are printed to the console. The resulting image with bounding boxes is displayed in a window using OpenCV's

```
# Function for drawing bounding boxes
def draw_box(image, box, class_name, score):
    x_min, y_min, x_max, y_max = map(int, box)
    color = np.random.randint(0, 255, size=3).tolist()  # Random color for each class
    cv2.rectangle(image, (x_min, y_min), (x_max, y_max), color, 2)
    label = f'{class_name}: {score:.2f}'
    cv2.putText(image, label, (x_min, y_min - 5), cv2.FONT_HERSHEY_SIMPLEX, 0.9, color, 2)
    return image

# Load the ONNX model
session = onnxruntime.InferenceSession(Model_PATH)
# Preprocess the image
original_image, input_tensor, original_size = preprocess_image(Image_PATH)
# Inference
outputs = session.run(None, {session.get_inputs()[0].name: input_tensor})
# Extract predictions
boxes, labels, scores = outputs[:3]
# Boxes should be a 1D array
boxes = boxes.flatten()
# Rescale boxes
scaled_boxes = rescale_boxes(boxes, original_size)
# Reshape scaled_boxes to be a 2D array where each row is a box
scaled_boxes = scaled_boxes.reshape(-1, 4)
# Filter detections using confidence threshold value
valid detections = scores.flatten() > Confidence THRESHOLD
```

Figure 15.6 "draw_box" function for creating bounding boxes and ONNX model inference.

```
# Check valid detections
if valid_detections.any():
    result_image = original_image.copy()
    for box, label, score in zip(scaled_boxes[valid_detections],
                                 labels.flatten()[valid_detections],
                                 scores.flatten()[valid_detections]):
        class_name = Class_NAMES.get(int(label), "Unknown")
        result_image = draw_box(result_image, box, class_name, score)
        print(f"Detected {class_name} with confidence {score:.2f}")
        print(f"Bounding box: {box}")

    # Display the result
    cv2.imshow('Object Detection Result', result_image)

    # Wait for a key press or window close
    while cv2.getWindowProperty('Object Detection Result', cv2.WND_PROP_VISIBLE) >= 1:
        keyCode = cv2.waitKey(50)
        if keyCode != -1:
            break
    cv2.destroyAllWindows()
else:
    print("Can't Detect")
# exit the program
sys.exit(0)
```

Figure 15.7 Verify valid detections and display output images with bounding boxes.

"imshow" function. The program waits for a key press or the user to close the window, and once that occurs, it closes the window and exits the program. If no valid detections are found, it prints "Can't Detect" and exits the program. These programming steps are shown in Figure 15.7.

To test the ONNX model, we need to execute the Python program from the "mydl" virtual environment by typing "workon mydl." From the terminal on Raspberry Pi 5, we should navigate to the "Desktop/pc2raspi" directory, where both the ONNX model and the test images are stored. The test images are random pictures of objects, mostly the ones we want to detect and classify: pencils, scissors, wrenches, and screwdrivers. Once in the "pc2raspi" directory, we run

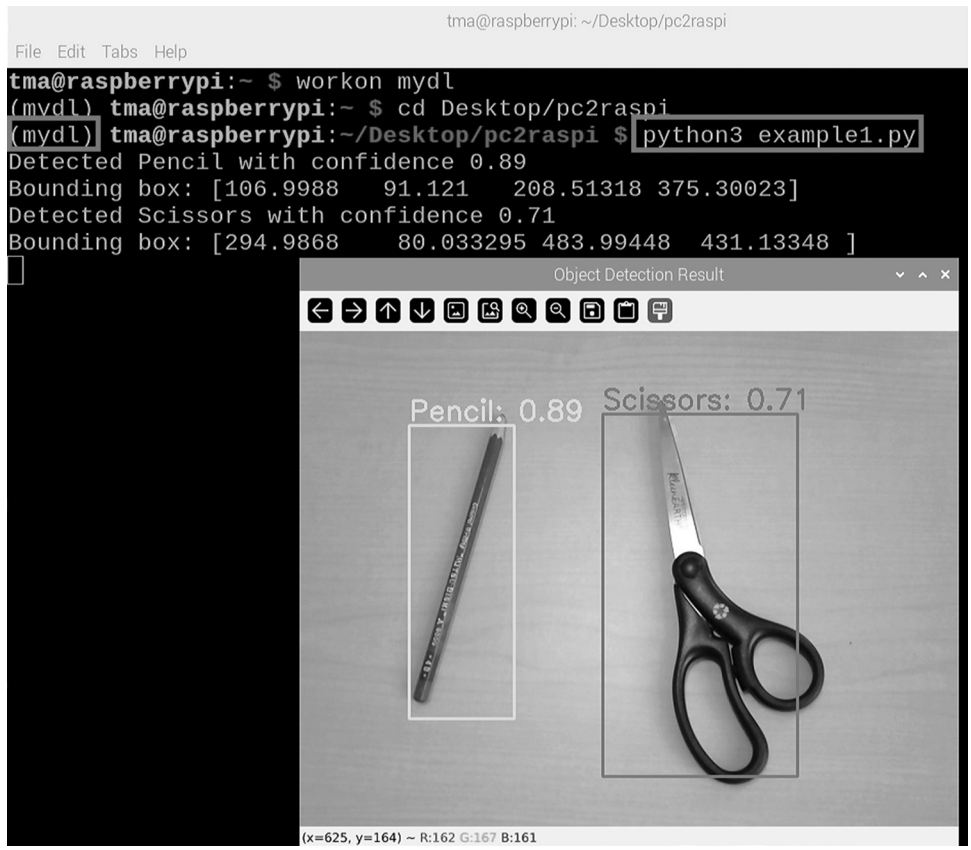


Figure 15.8 Model inference on a test image from the virtual environment "mydl."

the inference program using the command "python3 example1.py." This will generate predictions and draw bounding boxes around the detected objects. Figure 15.8 shows the inference on the "Test1.jpg" image in the virtual environment. We see that the model can detect both of the objects. If necessary, we may need to adjust the confidence threshold, especially if the images are captured in poor lighting conditions. The threshold can be fine-tuned below or above 70% based on other available objects and image quality.

Figure 15.9 presents the inference results of the ONNX model on four additional images (Test2.jpg, Test3.jpg, Test4.jpg, and Test5.jpg). In the Test4.jpg image, the model missed detecting another screwdriver. This outcome is influenced by the diversity of data used during training. To enhance the model's performance, adding more data with different variations of each class would be useful.

15.5 Conclusion

This chapter demonstrates the process of converting a custom-trained object detection with a classification model from PyTorch's ".pth" to ONNX format. The ONNX model is utilized for the model for inference on Raspberry Pi 5. While the PyTorch model can be saved as either a ".pth"

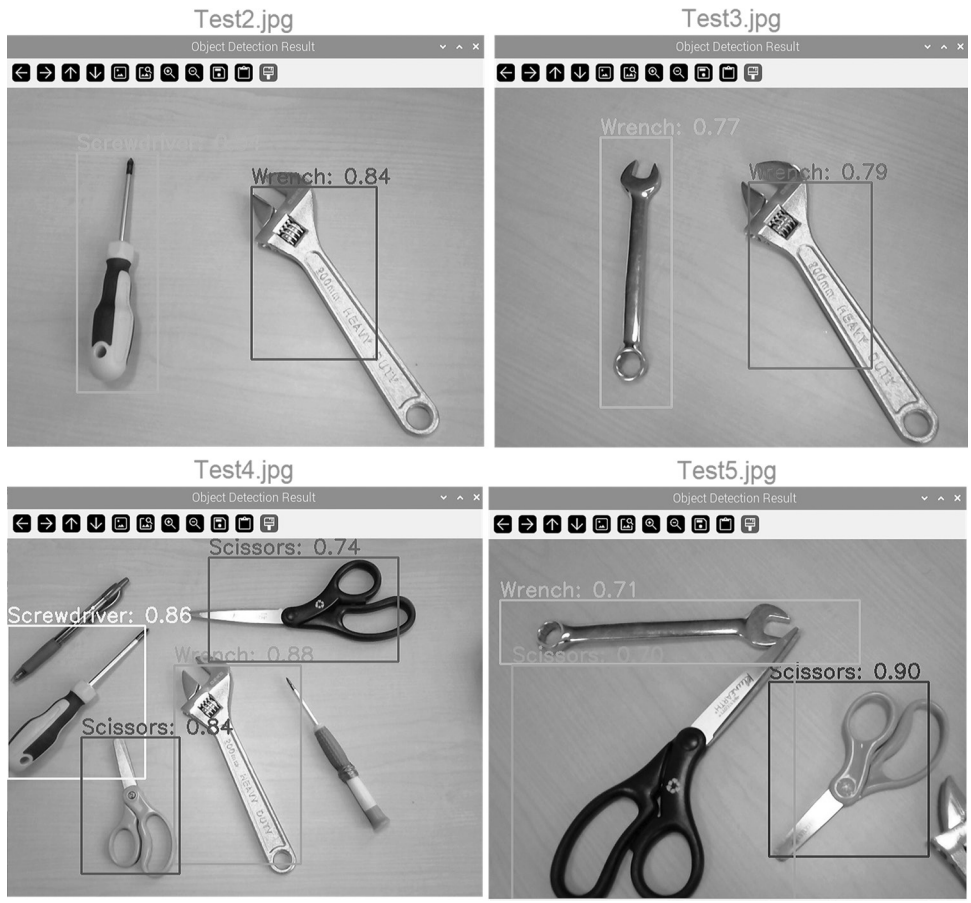


Figure 15.9 Model inference on test images.

or ".pt" file, the conversion process will be the same. The procedure outlined in this chapter is beneficial in situations where deploying a memory-intensive PyTorch model is not feasible due to portability, cost, and other constraints.

15.6 Exercise Problems

(Q1) Convert the PyTorch object detection model (best_fasterrcnn_model2.pth) trained on "dataset2" (from Chapter 10) into ONNX format for use on the Raspberry Pi 5.

(Q2) Use the ONNX model (best_fasterrcnn_model2.onnx) in Raspberry Pi and perform inference on the dice images given in Chapter 10.

For reference, review the "model_detection2_inference.py" file if needed. The outputs after inference on the Raspberry Pi should be similar to those shown in Figure 15.10.

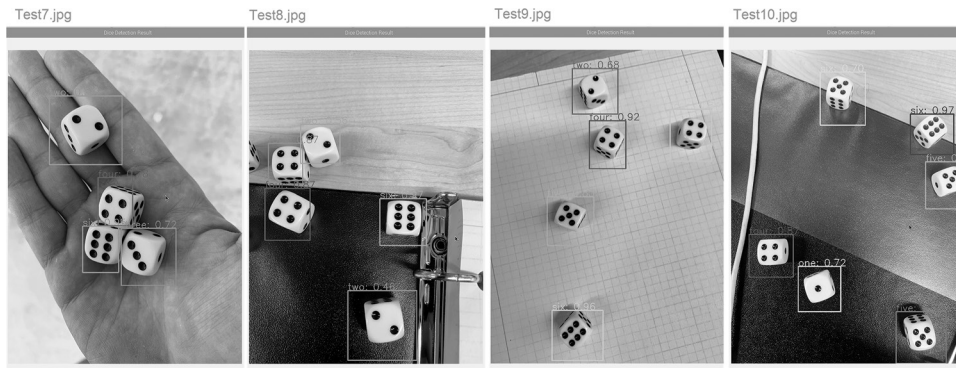


Figure 15.10 Inference on dice test images using the ONNX model on Raspberry Pi 5.

Reference

1 Runtime, O. (2024). *Accelerated Python – Machine Learning*. https://onnxruntime.ai.

Index

Note: Page numbers in *italics* and **bold** refers to figures and tables, respectively.

a

AArch64. *see* ARM 64-bit architecture (AArch64)
activation functions 15, 17, 22–23, 54
Adadelta optimizer 145
AdaGrad. *see* adaptive gradient algorithm (AdaGrad)
Adam optimizer 145, *146*
adaptive gradient algorithm (AdaGrad) 21, 145
adaptive learning rate techniques 24
Advanced Micro Devices (AMD) 7
AI. *see* artificial intelligence (AI)
Alexa 7
AlexNet 27, 32, 139
Amazon's Alexa 36
AMD. *see* Advanced Micro Devices (AMD)
Anaconda 41, 50
  "anaconda3" folder permission of user *46*
  installation and environment setup 42–44, *42–44*
  package manager 44
  Run Spyder using Anaconda Prompt and verify CUDA availability *46*
  for Windows system 41
"Anaconda Prompt" 44–45, 123–124, 133, 151
annotations 122, 151, 153, 154, 161
  defined 134
  directory 133
  setup data frame using 134–136
ANYmal C-legged robot 59
API. *see* application programming interface (API)
Apple's Siri 7, 36
application programming interface (API) 48
architecture-based hyperparameters 23
Arch Linux 85
ARM 64-bit architecture (AArch64) 109
artificial intelligence (AI) 1, 209, 225
  applications in robotics and image processing tasks 59
  deep learning and 7–9
audio datasets 121
"augmentor" method 137, *138*
automated hyperparameter optimization techniques 25
automatic data labeling 128
  of "captured_images" *130*
  in CSV files for range of images *130*
automatic image collection
  Python code in Code-oss *127*
  Python file in *vs.* code terminal and save images *129*
  using embedded device 127
  "while" loop for continuously capturing and saving frames *128*

b

backpropagation 18, 21–22, 27–28, 53–57
  algorithm 35
  in neural networks 27
backpropagation through time (BPTT) 34–35
backward propagation 18, 140
Bash. *see* Bourne Again SHell (Bash)
Basic Linear Algebra Subprograms (OpenBLAS) 198
batch 19, 24–25
  gradient descent 21
  of images 155–156
  normalization 139
  size 22–23, 134, 137, 141, 143–144, 151, 154, 226
Bayesian optimization 21, 25
BERT. *see* Bidirectional Encoder Representations from Transformers (BERT)
best validation loss (best_val_loss) 156
best_val_loss. *see* best validation loss (best_val_loss)
Bidirectional Encoder Representations from Transformers (BERT) 28
bidirectional recurrent neural networks (BiRNNs) 34–36
big data 7–9
binary cross-entropy loss function 19
BiRNNs. *see* bidirectional recurrent neural networks (BiRNNs)
"Bookworm" 191
Bourne Again SHell (Bash) 86
BPTT. *see* backpropagation through time (BPTT)
BypassAI 9

Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi, First Edition.
Tariq M. Arif.

© 2025 John Wiley & Sons, Inc. Published 2025 by John Wiley & Sons, Inc.

c

camera serial interface (CSI) 61
Canadian Institute For Advanced Research 5
CentOS 85, 104
Chainer 6
ChatGPT 28
CNNs. *see* convolutional neural networks (CNNs)
CNTK. *see* Microsoft Cognitive Toolkit (CNTK)
COCO. *see* common objects in context (COCO)
CodeProject 5
"collate_fn" function 153, *155*
common objects in context (COCO) 4, 214
compute unified device architecture (CUDA) 44, 133, 179
  installation 48
  NVIDIA 63, 86
compute unified device architecture deep neural network library (cuDNN) 48, 63, 78–79, 86, 107
convolutional neural networks (CNNs) 21, 23, 27, 28, 169, 211
  applications 36–37
  convolutional operations 28–29, *29, 30*
  fully connected layers 32
  historical background 27
  for multi-class car model classification problem *32*
  output layers 32
  overall architecture 32
  padding and stride 30–31
  pooling layers 31–32
convolutional operations 28–29, *29, 30*
Copyleaks 9
Coral Dev Board 2, **3**
Coral Dev Board Mini **3**
coral system-on-module (SoM) **3**
Coral USB Accelerator 2, **3**
Cortex-M Series processors 2
cost function 19–22, *20*, 54
cross-entropy loss 19
CSI. *see* camera serial interface (CSI)
CUDA. *see* compute unified device architecture (CUDA)
cuDNN. *see* compute unified device architecture deep neural network library (cuDNN)
cutting-edge AI technologies 5
"cv2.getTextSize()" function 174, 179
"cv2.imshow()" function 80, 172, 174, 218–219
"cv2.imwrite()" function 128
"cv2.kmeans" function 219
"cv2.putText" function 213, 216, 218, 229
"cv2.rectangle" function 179, 210, 213
CyclicLR 146
"Cython" 199

d

data augmentation 53, 131, 137, 152, *153*
data frame *136*, 151–152
  using annotations 134–136
data loader 141, 155–156, *155*
  and classification backbone 153–155
  and model configuration 137, *138*
data preprocessing and cleaning 130–131
data privacy and security in AI 9
datasets 4, 7, 9, 18, 121, 154, 169
  automatic data labeling 128–130
  automatic image collection using embedded device 127–128
  class and methods 136–137, *137*, 153
  data preprocessing and cleaning 130–131
  exercise problem 131
  manual dataset creation 122–127
  spatial 27
  types 121
Debian 85–86
  Debian 12, 191
  Debian-based distributions 104
decision trees 1
deep learning 1–2
  activation functions, role of 17
  adoptions in industry and business 8
  and artificial intelligence 7
  challenges and best practices for efficient tuning 25
  code 107
  computational hardware, advances in 7–8
  developer, and key features *6*
  essential packages for 50–52
  ethical, legal, and privacy concerns 8
  exercise problems 221–222
  face detection using HOG 209–210
  face landmark detection video *222*
  face recognition using transfer learning 211–214
  frameworks 5–6, 107
  future challenges 9
  gradient descent algorithm 19–21
  hyperparameter tuning 22–23
  learning types 18
  machine learning to 1
  models on Raspberry Pi 209
  neural networks 15
  neurons and layers 15–17
  real-time classification 217–218
  real-time object detection 214–217
  real-time segmentation 219–221
  training on network 18–19
  video frame objects *222*
  weight initialization and regularization 21–22
Deeplearning4j (DL4J) 6
deep neural network approach 1
deep reinforcement learning agent 24
deep transfer learning 146
  in embedded system 4–5
  tasks 150
"DefaultConfigs" class 134, *135*, 140–141, 151, 157–158
default configurations
  and libraries 134
  and random seeds 151, *152*
"DetectionDataset" class 153–154, *154*
dice face detections 165, *166*
DL4J. *see* Deeplearning4j (DL4J)
docker engine 107
  docker file, creation 116–118, *117*
  docker image, building 115

in embedded devices 107–108
exercise problems 119, *120*
host files in docker environment 110–114
image, building 116–118, *117*
install docker extension 116, *117*
Jetson inference docker 108–110
run Python through docker container 118–119
runtime image for MediaPipe ecosystem 115
"draw_boxes" function 164–165, *164*, 182, 185, 229, *230*
"draw_highest_score_box" function 160, *160*, 164
dropout 24–25, 53, 139, 178

e

"eduroam" network 71, 194
EfficientNet for optimized architectures 27
Elman Network 28
ELMo. *see* embeddings from language models (ELMo)
embedded devices 2, 5, 127, 146, 177–178
docker in 107–108
Jetson 59–62
low-powered 217
processing power and memory of common **3–4**
embedded systems 59, 85, 103, 115, 169, 187
application in 146
deep learning frameworks in 107
deep transfer learning in 4–5
Jetson Nano 59
modern 2–4
processing power and memory of common embedded devices **3–4**
embeddings from language models (ELMo) 28
epochs 23–25, 134, 140–141, 143–145, 151, 155–157, *158*
error function 19
Etcher software 63
"evaluate" function 156–157, *157*
"example1.py" file *112*, 210, *211*
"example1.py" script 113, 170
ExponentialLR 146

f

Facebook 5, 172
Facebook's AI Research lab (FAIR) 41, 198
face detection 210–211
dice 165, *166*
using HOG 209–210, *210*
real-time 115
"face_encodings" function 211
face recognition
tasks 23
using transfer learning 211–214, *212–214*
using video streams *214*
"while" loop for 212, *213*
facial recognition technology 9
FAIR. *see* Facebook's AI Research lab (FAIR)
Faster R-CNN 149–154, *150*, 158, 181
"get_model" function to create *159*
with ResNet-101 backbone 149
feature pyramid network (FPN) 150
Fedora 85, 104
fine tuning 17, 25, 142–144
training and validation losses and accuracies *144*
using different model 144–145
using different optimizers and schedulers 145–146
forward propagation 18
FPN. *see* feature pyramid network (FPN)
FPS. *see* frames per second (FPS)
frames per second (FPS) 217
fully connected layers 18, 27–28, 32, 54

g

gated recurrent units (GRUs) 34–35
Gemini 28
general public license (GPL) 85
general-purpose input/output (GPIO) 61, 187–188
generative pre-trained transformer (GPT) 28
genetic algorithms 21
"GetClassDesc()" function 174
"__getitem__" method 137, *137*, 153
"getLayerNames()" method 214
"get_model" function 154, *155*, 158, *159*
"get_valid_transform()" function 153
GitHub Discussions 5
Glorot initialization 21
Gluon 6
Google 5, 8, 115
Google Coral 2
devices 2, **3**
USB Accelerator 187
GoogLeNet 133, 138–139, *139*, 145, 147
GoogLeNet/Inception 27
model 109
pre-trained *110*
Google's Neural Machine Translation 36
Google's voice search 36
GPIO. *see* general-purpose input/output (GPIO)
GPL. *see* general public license (GPL)
GPT. *see* generative pre-trained transformer (GPT)
GPT-2 Output Detector 9
GPTZero 9
GPUs. *see* graphical/graphics processing units (GPUs)
gradient calculation in PyTorch 55–57, *55–57*, 139
gradient descent algorithm 19–21
graphical/graphics processing units (GPUs) 1, 24–25, 44, 140, 155, 178
grid search 25, 143
GRUs. *see* gated recurrent units (GRUs)

h

HDMI. *see* high-definition multimedia interface (HDMI)
high-definition multimedia interface (HDMI) 61
Hinge loss 19
histogram of oriented gradients (HOG) 209
face detection using 209–210
HOG. *see* histogram of oriented gradients (HOG)
Hopfield Network 28
host files in docker environment 110
attach host directory to root system and testing 113–114
create executable Python 111–113

Huber loss 19
hyperparameter(s) 20, 22, 143–144, 146
  architecture-based 23
  batch, iteration, and epochs 24–25
  dropout 24
  kernel size 23
  learning rate 23–24
  training-based 23
  tuning 22–23, 142
  weight decay 24

i

IBM. *see* International Business Machines Corporation (IBM)
IBM Watson Health 8
IDE. *see* integrated development environment (IDE); Interactive Development Environment (IDE)
ILSVRC. *see* ImageNet Large Scale Visual Recognition Challenge (ILSVRC)
image classification 18–19, 21, 23–24, 121–123, 130
  data loader and model configuration 137–140
  dataset class and methods 136–137
  default configurations and libraries 134
  embedded system, application in 146
  exercise problems 147
  fine tuning 142–146
  labels *142*
  model training 140–141
  problem statement 133
  setup data frame using annotations 134–136
  testing and inference 141–142, *142, 143*
  training and validation accuracy *141*
  training loop for classification model *140*
  training model for 133
  transfer learning on customized dataset *134*
  using ONNX 179–180
image datasets 121, 143
image file 63, 65, 75, 116, 124, 128, 137, 142
  import libraries *171*
  inference on 170–172
  OpenCV-compatible format *172*
ImageNet 4, 27, 171
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 133
image preprocessing 228–229, *229*
"imutils" 204, *204*
"Inception module" 133
Inception V3 139
inference 41. *see also* Jetson inference; model inference
  on image file 170–172
  on live video stream 173–175
  operation 5
  testing and 141–142
"__init__" method 136, *137*
integrated development environment (IDE) 133
Intel Movidius 2
Intel Neural Compute Stick 2, 2, **3**
Interactive Development Environment (IDE) 41
International Business Machines Corporation (IBM) 8
Internet of Things (IoT) 2, 60, 169, 225
intersection over union (IoU) 163–164
IoT. *see* Internet of Things (IoT)
IoU. *see* intersection over union (IoU)
iteration 24–25, 59, 140, 143, 157, 187, 219

j

Jasper 9
JetPack *76*, 86
  installation 62–63, *64*
  OpenCV in 75
Jetson AGX Xavier 2, **3**, 59
Jetson embedded devices 59
  vs. Code on Jetson 71–75
  direct setup 65
  exercise problems 81
  hardware and power requirements 61, *61–62*
  increase root partition size 65–68
  Jetpack installation 62–63, *64*
  Jetson Nano 60, *60*
  OpenCV and PyTorch in Jetson 75
  OpenCV library and test video capture functionality 79–80, *79–82*
  other settings 68–70
  setting up Jetson inference 76–78
  Wi-Fi driver 70–71
Jetson inference 77, 112, 174
  docker 108–110
  library 76, 179
Jetson Nano 2, **3**, 59, 60, *60*, 107–108, 127, 146, 169
  deep learning models on 169
  Desktop PC to 177
  exercise problems 176, 186
  GPU capabilities 75
  image file, inference on 170–172
  live video stream, inference on 173–175
  model training on PC 177
  ONNX model 172–173
  ONNX model inference 177–186
  power requirements 61
  pre-trained models 169–170
Jetson's embedded device 2
Jetson TX2 **3**, 59
Jetson Utils 174
Jetson Xavier NX 2, **3**, 59
John Deere 8
JPMorgan Chase 8

k

Kaiming He initialization 22
Keras 5, 17, 22, 41
kernel(s) 28–29, 87, 107
  filter 29, *30*
  Linux 85
  receptive fields 31
  size 23
K-means clustering, real-time segmentation using 219–221, *220*
"known_face_names" array 212

l

LabelImg 123, 130
  Anaconda Prompt *124*
  "labelImg-master. zip" download *123*
  opening directory *126*
  "predefined_classes. txt" file *125*
  Qt resource file *124*
  rectangular box creation *126*
  "resources. qrc" file *125*
  setup 123–127
Leaky ReLU 17
learning rate 20–21, 23–24
  scheduler 155, 157
  SGD with 155
learning types 18
LeCun initialization 22
LED. *see* light emitting diode (LED)
LeNet-5 27
"__len__" method 136, *137*
Levenberg–Marquardt algorithm 21
light emitting diode (LED) 65
Linux 63, 69, 70, 79, 85–86, *197*
  basic terminal commands and syntax 86–87
  common wildcard characters 95–96
  copy and move operations 92–93
  create, edit, and delete 90, *90–92*
  file system 87
  find, view, and get information 96–99
  install and uninstall packages using "sudo" 104, *105*
  navigating files and directories 87–89
  permission and ownership 99
  permissions using octal and symbolic notations 100–101
  Python code from terminal, creating and executing 93–94, *95*
  update permission using operators 101–104
live video stream, inference on 173–175, *174–175*
long short-term memory networks (LSTM networks) 28, 34, 35
loss dictionary (loss_dict) 156
loss functions 19, 143, 147
LSTM networks. *see* long short-term memory networks (LSTM networks)
Luxonis OAK-D cameras 2

m

machine learning 1, 8, 15, 63, 115, 121, 141, 172, 187, 225. *see also* deep learning
  algorithms 15, 221
  hyperparameter tuning in 22–23
  libraries 41
  optimization problems 20
  projects 62, 71
MAE. *see* mean absolute error (MAE)
make directory command (mkdir) 90
manual dataset creation 122
  classification and detection dataset 122–123
  LabelImg 123–127
many-to-many RNN architecture 33–34, *34*
many-to-one RNN architecture 33, *34*
mask R-CNN for image segmentation 27
Matplotlib 75, 141, 160, 164
mean absolute error (MAE) 19
mean squared error (MSE) 19, 147
MediaPipe 115, 119
  recognition functionality using *116*
  runtime image for MediaPipe ecosystem 115
memory storage technologies 7–8
microSD. *see* micro secure digital (microSD)
micro secure digital (microSD) 61, 191, *192, 194*
Microsoft 5, 8, 71
Microsoft Cognitive Toolkit (CNTK) 5
Microsoft Common Objects in Context 5
Microsoft's Cortana 36
micro universal serial bus (micro-USB) 61
micro-USB. *see* micro universal serial bus (micro-USB)
Mint 85
MIPI. *see* mobile industry processor interface (MIPI)
mkdir. *see* make directory command (mkdir)
MNASNet 139
MNIST. *see* Modified National Institute of Standards and Technology (MNIST)
mobile industry processor interface (MIPI) 61
MobileNet v2 139, *219*
model inference 142, 157
  with bounding boxes 229–231, *231, 232*
  import essential libraries and modules *159*
  for multiple object detection 163–165
  object detection with classification inference *161*
  preprocessing functions 158–160
  results on different test images *162*
  on test images 160
modern embedded systems 2–4
  processing power and memory of common embedded devices **3–4**
modern grammar-checking tools 9
Modified National Institute of Standards and Technology (MNIST) 4
MSE. *see* mean squared error (MSE)
multi-layer neural network 15, *16*
Multi Margin Loss function 147
multiple epochs 156–157, *158*
multiple object detection
  with classification 161–162, *162, 165*
  model inference for 163–165
MultistepLR 146
MXNet 6, 41
MyCobot 280, 60

n

Negative Log-Likelihood Loss 19
"Neocognitron" 27
"net.Classify()" method 179
neural networks 1, 15, 19, 24, 53
neuromorphic computing 8
neurons and layers 15–17, *16, 17*
NGC. *see* NVIDIA GPU Cloud (NGC)
NMS. *see* non-maximum suppression (NMS)

non-maximum suppression (NMS) 163, 182, *183, 185*
"non_max_suppression" function 163–164, *163*, 182
no padding. *see* valid padding
normalization 54, 131, 136, 170
NumPy 5, 50, 53, 77, 107, 115, 134, 151, 173–174, 197, 203, 217, 228
NVIDIA GPU Cloud (NGC) 108
NVIDIA Jetson devices 2
NVIDIA Jetson series 2, **3–4**

o

object detection 23, 79, *114*, 121, 149. *see also* real-time object detection
  "albumentations" library for anaconda *150*
  data frame and process labels 151–152, *152*
  data loader and classification backbone 153–155
  dataset class and methods 153
  default configurations and random seeds 151
  "draw_boxes" function *184*
  exercise problems 167
  import modules and libraries 149–151
  model inference 157–160, 163–165
  multiple epochs 156–157
  multiple object detection with classification 161–162
  training and validation approach 155–156
  training process of *150*
  transformers, training and validation of 152–153
  using ONNX 181–186, *183–186*
  using YOLOv3 214–217
objective function 19
OCR. *see* optical character recognition (OCR)
one-to-many RNN architecture 33, *33*
ONNX model. *see* open neural network exchange model (ONNX model)
OpenBLAS. *see* Basic Linear Algebra Subprograms (OpenBLAS)
OpenCV. *see* Open Source Computer Vision (OpenCV)
open neural network exchange model (ONNX model) 6, 172–173, *174*, 177, 201, 225
  bounding boxes, model inference with 229–231, *230–232*
  continuous while loop for processing frames *180*
  convert PyTorch Model to 172–173
  image classification using ONNX 179–180, *179*
  image preprocessing 228–229, *229*
  import libraries and define configurations 228, *228*
  inference using Jetson inference library *176*
  model inference 177–186
  object detection using 181–186
  PyTorch's.pth to 181, *182*, 225–227
  PyTorch's.pt to ONNX conversion 177–178, *178*
  Raspberry Pi 5, model inference on 228–231
  real-time inference using *180*
  Runtime 225–226, *226*
Open Source Computer Vision (OpenCV) 4, 50, 63, 65, 75, 79, 173–174, 182, 216–218, 226, 228
  BGR color space for 219
  "cvtColor" function 174
  installation 198–200
  in Jetson 75
  "VideoCapture" function 174
operating systems (OS) 48, 61, 63, 79, 86–87
  Linux-based 85
  Python's "os" for 134
  setup 190–197
optical character recognition (OCR) 204–205
Originality 9
OS. *see* operating systems (OS)

p

padding 30–31, 53
particle swarm optimization (PSO) 21
Phrasly.ai 9
PIL. *see* Pillow (PIL); Python Imaging Library (PIL)
Pillow (PIL) 201, *201*
PolynomialLR 146
pooling layers 28, 31–32
"predicted" class index 171, *171*
"preprocess_image" function 158–160, 182, 228–229, *229*
preprocessing 25
  data 130–131
  functions 158–160
  image 228
  techniques 131
  transformations 152, *153*
pre-trained models 51, *51*, 76, 139, 144, 146, 150, 169–170, 209, 218
  download and test 109
  on large datasets 4
privacy concerns 8, 225
probabilistic methods 21
process labels 151–152
product recommendation system 7
PSO. *see* particle swarm optimization (PSO)
"pyqt" 124
Python 71, *74, 78*, 213
  code 5
  compilers and interpreters 42
  create and execute Python code from Linux terminal 93–94, *95*
  dependencies 218
  program in *vs.* Code 74
  python3 203, 226
  Spyder IDE for 41
  steps for installing *73*
Python Imaging Library (PIL) 201
PyTorch 1, 5, 6, 22, 41, 144–145, 169
  Anaconda and PyTorch for Windows system 41
  Anaconda installation and environment setup 42–44, *42–44*
  CUDA and cuDNN installation 48, *48–50*
  exercise problems 57
  gradient calculation in 55–57
  installation 198–200
  installation command for *45*
  in Jetson 75
  models 171–172
  to ONNX 172–173
  quantized model, classification using 217–218, *217*
  run Spyder using Anaconda Prompt *46*
  setting up PyTorch framework 44
  tensors in 53

torch operations in 53–54, *54*
Visual Studio Professional 47, *47*
PyTorch's.pth to ONNX 225
conversion 181
model conversion in Raspberry Pi 5, 226–227, *227*
ONNX Runtime 225–226, *226*
PyTorch's.pt to ONNX conversion 177–178

q

QNNPACK. *see* quantized neural network package (QNNPACK)
"qtbase5-dev" development package 203
quantized neural network package (QNNPACK) 217–218
quantum computing 8
QuillBot 9
Quora 5

r

random search 25
random seeds 151, *152*
random tensors 53–54, *54*
Raspberry Pi 2, **4**, 107–108, 127, 146, 187, 225
"cmake" "libgtk-3-dev," "libboost-all-dev," and "python3-dev" 201, *202*
direct setup 190
essential packages 201–205, *201–206*
exercise problems 207, 232–233
flashing OS image 191–194, *192–193*
hardware and power requirements 188–190
"imutils" and "tesseract-ocr" 204–205, *204*
Model B 187
model training on Desktop PC 225
"numpy" library 203, *204*
"onnx" and "onnxscript" 201, *202*
ONNX model inference on Raspberry Pi 5, 228–231
operating system setup 190–197
"pip" and "pillow" 201, *201*
PyTorch and OpenCV installation 198–200, *200*
PyTorch's.pth to ONNX 225–227
Qt "xcb" platform 205, *205–206*
setting up OS 194–197, *194–197*
virtual environment 197–198
"wheel," "dlib," and "facerecognition" 202, *203*
Wi-Fi network and internet connection in *196*
workon mydl 198, *199*
Raspberry Pi 4 Model B **4**
Raspberry Pi 5, 4, 86, 187–188
"Choose Device" from *193*
hardware components used for **189–190**
with heatsink mounted blower fan *189*
inference on dice test images using ONNX model on *233*
key components and peripheral interfaces *188*
model conversion in 226–227, *227*
ONNX model inference on 228–231
peripheral connections to *191*
Raspberry Pi Imager 191
"Choose Device" from *193*
"Choose OS" from *193*
for Windows computer *192*
Raspbian 190
"read_images" method 137, *138*
real-time classification of deep-learning, using PyTorch's quantized model 217–218, *217–219*
real-time object detection 27, 110, 112, 146, 214, *215*
bounding boxes, creating *216*
logic for creating bounding boxes *215*
using YOLOv3 214–217, *215, 216*
real-time segmentation 219
convert segmented frame *220*
import libraries, initialize video stream, and use "while" loop *220*
original, segmented, and overlay frames *221*
using K-means clustering 219–221, *220*
Rectified Linear Unit (ReLU) 17
recurrent neural networks (RNNs), 21, 27, 169. *see also* convolutional neural networks (CNNs)
applications 36–37
basic architectures 33–36
framework 33
historical background 28
Reddit 5
Red Hat 85
region proposal network (RPN) 149
reinforcement learning 18, 121
".release()" method 128, 175, 180
ReLU. *see* Rectified Linear Unit (ReLU)
"rescale_boxes" function 159–160, 228–229, *229*
ResearchGate 5
Residual Networks (ResNet) 27, 139, 145
"resize_for_display" function 182, 185
ResNet. *see* Residual Networks (ResNet)
ResNet-18 model 169–170, *170*, 172
"resnet18.onnx" model 173
Resnet50 model 144–145, *145*
ResNet-101 149–150, 153–155, 158, 181
ResNeXt 139
"Revo Uninstaller Pro" 45
RMSprop. *see* root mean square propagation (RMSprop)
RNNs. *see* recurrent neural networks (RNNs)
robotics 1–2, 5, 36, 59, 62, 146, 169, 187
root mean square propagation (RMSprop) 21, 24, 145
RPN. *see* region proposal network (RPN)

s

Sapling 9
scaling 131
SciPy 5
SDK. *see* software development kit (SDK)
SD Memory Card Formatter 63, *64*
secure shell (SSH) 194
semi-supervised datasets 121
semi-supervised learning 18
"set_all_random_seed" function 151
SGD optimizer. *see* stochastic gradient descent optimizer (SGD optimizer)
Siemens 8
sigmoid function 16–17
silicon photonics technology 8
simple recurrent neural network (SRNN) 28
single shot multibox detector (SSD) 110

software development kit (SDK) 62
SoM. *see* coral system-on-module (SoM)
Sonnet 6
speech recognition 2, 18, 28, 33, 35–36, 121
Spyder integrated development environment 133
SqueezeNet 139
SRNN. *see* simple recurrent neural network (SRNN)
SSD. *see* single shot multibox detector (SSD)
SSH. *see* secure shell (SSH)
Stack Exchange 5
Stack Overflow 5
Stealth AI 9
StealthGPT 9
StealthWriter 9
"StepLR" scheduler 146, *147*
stochastic gradient descent optimizer (SGD optimizer) 55, 145, 155
stride 8, 30–31
"sudo," install and uninstall packages using 104, *105*
"sudo apt-get update" command 195
supervised datasets 121
supervised learning 18
support vector machines (SVM) 1, 32, 209
SVM. *see* support vector machines (SVM)
swarm intelligence techniques 21

t

tensor(s) 52
  2D with 3 by 3 size using PyTorch *53*
  3D with 3 by 1 by 3 size using PyTorch *53*
  defining 52
  element-wise division and matrix multiplication of *54*
  in PyTorch 53
  random 53–54, *54*
  zero 53
"TensorBoard" 6
TensorFlow 1, 5–6, 22, 41, 169, 197, 226
TensorFlow Lite 187
tensor processing units (TPUs) 1
TensorRT 63, 79, 86, 172–173, 177
Tesla 7–8
"tesseract-ocr" 204–205, *204*
TextCortex 9
text datasets 121
Theano 6
Thonny IDE 209, *210*
"torchaudio" libraries 199
"torch.onnx.export()" function 173, 178, 226–227
torch operations in PyTorch 53–54, *54*
TorchScript model 218, *218*
"torchvision" library 139, 169, 199, 226
"torchvision. models. detection" module 150
TPUs. *see* tensor processing units (TPUs)
traditional machine learning methods 1
trained AI detection tools 9
training and validation approach 155–156
training-based hyperparameters 23
"train_one_epoch" function 155, *156*, 157
transfer learning 4, 25, 206
  approach for real-time object detection 214
  face recognition using 211–214
transformers
  training and validation of 152–153
  transformer-based models 28
truncated normal distribution 22
Turnitin 9

u

Ubuntu 63, 70, 85–87, 104, 191
U-Net for biomedical imaging 27
unsupervised datasets 121
unsupervised learning 18

v

valid padding 30–31
VGG 139
VGGNet 27
video datasets 121
virtual assistant technologies 36
virtual environment 197–198, *198*
virtual network computing (VNC) 194
Vision Transformers 27
visual studio (VS) 179
  code 71
  on Jetson 71–75
Visual Studio Professional 47–48, *47*
VNC. *see* virtual network computing (VNC)
VS. *see* visual studio (VS)

w

weight decay 24–25, 155
weight initialization and regularization technique 21–22
"while" loop 210, 212, 216, 219
  for continuously capturing and saving frames *128, 213*
  import libraries, initialize video stream, and use *220*
  process model output in *218*
  PyTorch's Quantized Model and *218*
  real-time object detection *215*
Wide ResNet 139
Wi-Fi driver 65, 70–71
wildcard characters 95–96
Wordtune 9

x

Xavier Glorot initialization 21–22
Xilinx Zynq UltraScale+ MPSoC 4

y

YOLO for real-time object detection 27
YOLOv3. *see* you only look once, version 3 (YOLOv3)
you only look once, version 3 (YOLOv3) 214, 217
  configuration, weights, and COCO label names **214**, *215*
  object detection using 214–217

z

ZeroGPT 9
zero padding 30–31
zero tensors 53
"zip" function 153