MACHINE LEARNING MASTERY

Second Edition

Deep Learning with Python

Develop Deep Learning Models with TensorFlow and Keras

Authors

Jason Brownlee

Adrian Tam

Zhe Ming Chng

Disclaimer

Credits

Authors: Jason Brownlee, Adrian Tam, Zhe Ming Chng
Lead Editor: Adrian Tam
Technical Reviewers: Darci Heikkinen, Jerry Yiu, Amy Lam

Copyright

Edition: v2.00

Contents

Preface to First Edition

Deep learning is a fascinating field. Artificial neural networks have been around for a long time, but something special has happened in recent years. The mixture of new faster hardware, new techniques and highly optimized open source libraries allow very large networks to be created with frightening ease.

This new wave of much larger and much deeper neural networks are also impressively skillful on a range of problems. I have watched over recent years as they tackle and handily become state-of-the-art across a range of difficult problem domains. Not least object recognition, speech recognition, sentiment classification, translation and more.

When a technique comes along that does so well on such a broad set of problems, you have to pay attention. The problem is where do you start with deep learning? I created this book because I thought that there was no gentle way for Python machine learning practitioners to quickly get started developing deep learning models.

In developing the lessons in this book, I chose the best of breed Python deep learning library called Keras that abstracted away all of the complexity, ruthlessly leaving you an API containing only what you need to know to efficiently develop and evaluate neural network models.

This is the guide that I wish I had when I started apply deep learning to machine learning problems. I hope that you find it useful on your own projects and have as much fun applying deep learning as I did in creating this book for you.

Jason Brownlee
Melbourne, Australia
2016

Preface to Second Edition

Deep learning is evolving fast, so are the deep learning libraries. If we compare the tools we have in 2016 and 2022, there is a fascinating change in mere six years.

When the first edition of this book was written, Keras and TensorFlow were separate libraries. TensorFlow at the time was unintuitive and difficult to use. Nowadays, they are combined into one and the other backends for Keras such as Theano ceased development. The eager execution syntax of TensorFlow also brings it to a new age. Therefore, we need to bring this book up to date with the new code that can run in the modern version of Keras.

Same as the first edition, we try to cover a broad topics in deep learning without going too deep. We covered enough to help you see why deep learning models are impressive. This book is intended for practitioners and therefore each chapter is a gentle introduction to an idea in deep learning without distracting you with too much theory and mathematics. We also enriched the first edition with some more examples and tools, thanks to the new features included in the libraries over the past few years.

As in the first edition, we wish this book can help you start applying deep learning quickly. It is a book with examples that you can copy. By doing so, you should have a jump-start to use TensorFlow and Keras library for some practical problems. This can be a stepping stone to something bigger, and your journey starts here!

Jason Brownlee
Melbourne, Australia

Zhe Ming Chng
Singapore

Adrian Tam
New York, U.S.A.

2022

Introduction

Welcome to Deep Learning with Python, Second Edition. This book is your guide to deep learning in Python. You will discover the Keras library in TensorFlow for deep learning and how to use it to develop and evaluate deep learning models. In this book you will discover the techniques, recipes and skills in deep learning that you can then bring to your own machine learning projects.

Deep learning does have a lot of fascinating math under the covers, but you do not need to know it to be able to pick it up as a tool and wield it on important projects and deliver real value. From the applied perspective, deep learning is quite a shallow field and a motivated developer can quickly pick it up and start making very real and impactful contributions. This is our goal for you and this book is your ticket to that outcome.

Deep Learning the Wrong Way

If you ask a deep learning practitioner how to get started with neural networks and deep learning, what do they say? They say things like

- ▷ You must have a strong foundation in linear algebra.
- ▷ You must have a deep knowledge of traditional neural network techniques.
- ▷ You really must know about probability and statistics.
- ▷ You should really have a deep knowledge of machine learning.
- ▷ You probably need to be a PhD in computer science.
- ▷ You probably need 10 years of experience as a machine learning developer.

You can see that the *common sense* advice means that it is not until after you have completed years of study and experience that you are ready to actually start developing and evaluating machine learning model for your machine learning projects.

We think this advice is dead wrong.

Deep Learning with Python

The approach taken with this book and with all of Machine Learning Mastery is to flip the traditional approach. If you are interested in deep learning, start by developing and evaluating deep learning models. Then if you discover you really like it or have a knack for it, later you can step deeper and deeper into the background and theory, as you need it in order to serve you in developing better and more valuable results. This book is your ticket to jumping in and making a ruckus with deep learning.

We have used many of the top deep learning platforms and libraries. We chose what we think is the best-of-breed platform for getting started, and very quickly developing powerful and even state-of-the-art deep learning models in the Keras deep learning library for Python. Unlike R, Python is a fully featured programming language allowing you to use the same libraries and code for model development as you can use in production. Unlike Java, Python has the SciPy stack for scientific computing and scikit-learn which is a professional grade machine learning library.

The top numerical platforms for developing deep learning models is TensorFlow developed at Google. It is developed for use in Python and the Keras library is built-in. Keras wraps the numerical computing complexity of TensorFlow providing a concise API that we will use to develop our own neural network and deep learning models.

You will develop your own and perhaps your first neural network and deep learning models while working through this book. You will have the skills to bring this amazing new technology to your own projects. It is going to be a fun journey and we can't wait to start.

Book Organization

There are three kinds of chapters in this book.

- ▷ **Lessons**, where you learn about specific features of neural network models and how to use specific aspects of the Keras API.
- ▷ **Projects**, where you will pull together multiple lessons into an end-to-end project and deliver a result, providing a template for your own projects.
- ▷ **Recipes**, where you can copy and paste the standalone code into your own project, including all of the code presented in this book.

Lessons and Projects

Lessons are discrete and are focused on one topic, designed for you to complete in one sitting. You can take as long as you need, from 20 minutes if you are racing through, to hours if you want to experiment with the code or ideas and improve upon the presented results. Your lessons are divided into five parts:

- ▷ Background
- ▷ Multilayer Perceptrons
- ▷ Advanced Multilayer Perceptrons and Keras

- ▷ Convolutional Neural Networks
- ▷ Recurrent Neural Networks

Part 1: Background

In this part you will learn about the TensorFlow and Keras libraries that lay the foundation for your deep learning journey. This part of the book includes the following lessons:

- ▷ Overview of Some Deep Learning Libraries
- ▷ Introduction to TensorFlow
- ▷ Using Autograd in TensorFlow to Solve a Regression Problem
- ▷ Introduction to the Keras

The lessons will introduce you to the important foundational libraries that you need to install and use on your workstation. You will also learn about the relationship between TensorFlow and Keras. At the end of this part you will be ready to start developing models in Keras on your workstation.

Part 2: Multilayer Perceptrons

In this part you will learn about feedforward neural networks that may be deep or not and how to expertly develop your own networks and evaluate them efficiently using Keras. This part of the book includes the following lessons:

- ▷ Understanding Multilayer Perceptrons
- ▷ Building Multilayer Perceptron Models in Keras
- ▷ Develop Your First Neural Network with Keras
- ▷ Evaluate the Performance of Deep Learning Models
- ▷ Three Ways to Build Models in Keras

These important lessons are tied together with three foundation projects. These projects demonstrate how you can quickly and efficiently develop neural network models for tabular data and provide project templates that you can use on your own regression and classification machine learning problems. These projects include:

- ▷ Project: Multiclass Classification of Iris Species
- ▷ Project: Binary Classification of Sonar Returns
- ▷ Project: Regression Problem of Boston House Prices

At the end of this part you will be ready to discover the finer points of deep learning using the Keras API.

Part 3: Advanced Multilayer Perceptrons

In this part you will learn about some of the more finer points of the Keras library and API for practical machine learning projects and some of the more important developments in applied neural networks that you need to know in order to deliver world class results. This part of the book includes the following lessons:

- ▷ Use Keras Deep Leraning Models with scikit-learn
- ▷ How to Grid Search Hyperparameters for Deep Learning Models
- ▷ Save and Load Your Keras Model with Serialization
- ▷ Keep the Best Models During Training with Checkpointing
- ▷ Understand Model Behavior During Training by Plotting History
- ▷ Using Activation Functions in Neural Networks
- ▷ Loss Functions in TensorFlow
- ▷ Reduce Overfitting with Dropout Regularization
- ▷ Lift Performance with Learning Rate Schedules
- ▷ Introduciton to tf.data API

At the end of this part you will know how to confidently wield Keras on your machine learning projects with a focus on the finer points of investigating model performance, persisting models for later use and gaining lifts in performance over baseline models.

Part 4: Convolutional Neural Networks

In this part you will receive a crash course in the dominant model for computer vision machine learning problems and some natural language problems and learn how you can best exploit the capabilities of the Keras API for your own projects. This part of the book includes the following lessons:

- ▷ Crash Course in Convolutional Neural Networks
- ▷ Understanding the Design of a Convolutional Neural Network
- ▷ Improve Model Performance with Image Augmentation
- ▷ Image Augmentation with Keras Preprocessing Layers and t f.image

The best way to learn about this impressive type of neural network model is to apply it. You will work through three larger projects and apply CNN to image data for object recognition and text data for sentiment classification.

- ▷ Project: Handwritten Digit Recognition
- ▷ Project: Object Recognition in Photographs
- ▷ Project: Predict Sentiment from Movie Reviews

After completing the lessons and projects in this part, you will have the skills and the confidence to use the templates and recipes to tackle your own deep learning projects using convolutional neural networks.

Part 5: Recurrent Neural Networks

In this part you will receive a crash course in the dominant model for data with a sequence or time component and learn how you can best exploit the capabilities of the Keras API for your own projects. This part of the book includes the following lessons:

- ▷ Crash Course In Recurrent Neural Networks
- ▷ Time Series Predictions with Multilayer Perceptrons
- ▷ Time Series Predictions with LSTM Networks
- ▷ Understanding Stateful LSTM Recurrent Neural Networks

The best way to learn about this complex type of neural network model is to apply it. You will work through two larger projects and apply RNN to sequence classification and text generation.

- ▷ Project: Sequence Classification of Movie Reviews.
- ▷ Project: Text Generation with Alice in the Wonderland.

After completing the lessons and projects in this part, you will have the skills and the confidence to use the templates and recipes to tackle your own deep learning projects using recurrent neural networks.

Conclusions

The book concludes with some resources that you can use to learn more information about a specific topic or find help if you need it, as you start to develop and evaluate your own deep learning models.

Recipes

Building up a catalog of code recipes is an important part of your deep learning journey. Each time you learn about a new technique or new problem type, you should write up a short code recipe that demonstrates it. This will give you a starting point to use on your next deep learning or machine learning project.

As part of this book you will receive a catalog of deep learning recipes. This includes recipes for all of the lessons presented in this book, as well as complete code for all of the projects. You are strongly encouraged to add to and build upon this catalog of recipes as you expand your use and knowledge of deep learning in Python.

Requirements for This Book

Python and SciPy

You do not need to be a Python expert, but it would be helpful if you know how to install and setup Python and SciPy. The lessons and projects assume that you have a Python and SciPy environment available. This may be on your workstation or laptop, it may be in a VM or a Docker instance that you run, or it may be a server instance that you can configure in the cloud. You will be guided as to how to install the deep learning libraries TensorFlow and Keras in Part I of the book. If you have trouble, you can follow the step-by-step tutorial in Appendix B.

Machine Learning

You do not need to be a machine learning expert, but it would be helpful if you knew how to navigate a small machine learning problem using scikit-learn. Basic concepts like cross-validation and one-hot encoding used in lessons and projects are described, but only briefly. There are resources to go into these topics in more detail at the end of the book, but some knowledge of these areas might make things easier for you.

Deep Learning

You do not need to know the math and theory of deep learning algorithms, but it would be helpful to have some basic ideas of the field. You will get a crash course in neural network terminology and models, but we will not go into much detail. Again, there will be resources for more information at the end of the book, but it might be helpful if you can start with some ideas about neural networks.

Note: All tutorials can be completed on standard workstation hardware with a CPU. GPU is not required. Some tutorials later in the book can be speed up significantly by running on the GPU and a suggestion is provided to consider using GPU hardware at the beginning of those sections. You can access GPU hardware easily and cheaply in the cloud and a step-by-step procedure is taught on how to do this in Appendix C.

Your Outcomes from Reading This Book

This book will lead you from being a developer who is interested in deep learning with Python to a developer who has the resources and capabilities to work through a new dataset end-to-end using Python and develop accurate deep learning models. Specifically, you will know:

- ▷ How to develop and evaluate neural network models end-to-end.
- ▷ How to use more advanced techniques required for developing state-of-the-art deep learning models.
- ▷ How to build larger models for image and text data.
- ▷ How to use advanced image augmentation techniques in order to lift model performance.
- ▷ How to get help with deep learning in Python.

From here you can start to dive into the specifics of the functions, techniques and algorithms used with the goal of learning how to use them better in order to deliver more accurate predictive models, more reliably in less time. There are a few ways you can read this book. You can dip into the lessons and projects as your need or interests motivate you. Alternatively, you can work through the book end-to-end and take advantage of how the lessons and projects built toward complexity and range. We recommend the latter approach.

To get the very most from this book, we recommend taking each lesson and project and build upon them. Attempt to improve the results, apply the method to a similar but different problem, and so on. Write up what you tried or learned and share it on your blog, social media or send us an email at `jason@MachineLearningMastery.com`. This book is really what you make of it and by putting in a little extra, you can quickly become a true force in applied deep learning.

What This Book Is Not

This book solves a specific problem of getting you, a developer, up to speed applying deep learning to your own machine learning projects in Python. As such, this book was not intended to be everything to everyone and it is very important to calibrate your expectations. Specifically:

- ▷ **This is not a deep learning textbook**. We will not be getting into the basic theory of artificial neural networks or deep learning algorithms. You are also expected to have some familiarity with machine learning basics, or be able to pick them up yourself.
- ▷ **This is not an algorithm book**. We will not be working through the details of how specific deep learning algorithms work. You are expected to have some basic knowledge of deep learning algorithms or be able to pick up this knowledge yourself.
- ▷ **This is not a Python programming book**. We will not be spending a lot of time on Python syntax and programming (e.g., basic programming tasks in Python). You are expected to already be familiar with Python or be a developer who can pick up a new C-like language relatively quickly.

You can still get a lot out of this book if you are weak in one or two of these areas, but you may struggle picking up the language or require some more explanation of the techniques. If this is the case, see the Getting More Help chapter at the end of the book and seek out a good companion reference text.

Summary

It is a special time right now. The tools for applied deep learning have never been so good. The pace of change with neural networks and deep learning feels like it has never been so fast, spurred by the amazing results that the methods are showing in such a broad range of fields. This is the start of your journey into deep learning and we are excited for you. Take your time, have fun and we're so excited to see where you can take this amazing new technology to.

Next

Let's dive in. Next up is Part I where you will take a whirlwind tour of the foundation libraries for deep learning in Python, namely the numerical library TensorFlow and the library you will be using throughout this book called Keras.

Background I

1 Overview of Some Deep Learning Libraries

Machine learning is a broad topic. Deep learning, in particular, is a way of using neural networks for machine learning. A neural network is probably a concept older than machine learning, dating back to the 1950s. Unsurprisingly, there were many libraries created for it.

The following aims to give an overview of some of the famous libraries for neural networks and deep learning. After finishing this chapter, you will learn

- ▷ Some of the deep learning or neural network libraries
- ▷ The functional difference between two common libraries, PyTorch and TensorFlow

Let's get started.

Overview

This chapter is in three parts; they are:

- ▷ The C++ Libraries
- ▷ Python Libraries
- ▷ PyTorch and TensorFlow

1.1 The C++ Libraries

Deep learning has gained attention in the last decade. Before that, there was little confident in how to train a neural network with many layers. However, understanding how to build a multilayer perceptron was around for many years.

Before we had deep learning, probably the most famous neural network library was `libann`. It is a library for C++, and the functionality is limited due to its age. This library has since stopped development. A newer library for C++ is OpenNN, which allows modern C++ syntax.

But that's pretty much all for C++. The rigid syntax of C++ may be why we do not have too many libraries for deep learning. The training phase of a deep learning project is about experiments. We want some tools that allow us to iterate faster. Hence a dynamic programming language could be a better fit. Therefore, you see Python come on the scene.

1.2 Python Libraries

One of the earliest libraries for deep learning is Caffe. It was developed at U.C. Berkeley specifically for computer vision problems. While it is developed in C++, it serves as a library with a Python interface. Hence we can build our project in Python with the network defined in a JSON-like syntax.

Chainer is another library in Python. It is an influential one because the syntax makes a lot of sense. While it is less common nowadays, the API in Keras and PyTorch bears a resemblence to Chainer. The following is an example from Chainer's documentation, and you may mistake it as Keras or PyTorch:

```python
import chainer
import chainer.functions as F
import chainer.links as L
from chainer import iterators, optimizer, training, Chain
from chainer.datasets import mnist

train, test = mnist.get_mnist()
batchsize = 128
max_epoch = 10

train_iter = iterators.SerialIterator(train, batchsize)

class MLP(Chain):
    def __init__(self, n_mid_units=100, n_out=10):
        super(MLP, self).__init__()
        with self.init_scope():
            self.l1 = L.Linear(None, n_mid_units)
            self.l2 = L.Linear(None, n_mid_units)
            self.l3 = L.Linear(None, n_out)

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

# create model
model = MLP()
model = L.Classifier(model)  # using softmax cross entropy

# set up optimizer
optimizer = optimizers.MomentumSGD()
optimizer.setup(model)

# connect train iterator and optimizer to an updater
updater = training.updaters.StandardUpdater(train_iter, optimizer)
```

```
# set up trainer and run
trainer = training.Trainer(updater, (max_epoch, 'epoch'), out='mnist_result')
trainer.run()
```

Listing 1.1: Example Chainer code

The other obsoleted library is Theano. It has ceased development, but once upon a time, it was a major library for deep learning. The earlier version of the Keras library allows you to choose between a Theano or TensorFlow backend. Indeed, neither Theano nor TensorFlow are deep learning libraries precisely. Rather, they are tensor libraries that make matrix operations and differentiation handy, upon where deep learning operations can be built. Hence these two are considered replacements for each other from Keras' perspective.

CNTK from Microsoft and Apache MXNet are two other libraries that worth mentioning. They are large with interfaces for multiple languages. Python, of course, is one of them. CNTK has C♯ and C++ interfaces, while MXNet provides interfaces for Java, Scala, R, Julia, C++, Clojure, and Perl. But recently, Microsoft decided to stop developing CNTK. MXNet does have some momentum, and it is probably the most popular library after TensorFlow and PyTorch.

Below is an example of using MXNet via the R interface. Conceptually, you see the syntax is similar to Keras' functional API:

```
require(mxnet)

train <- read.csv('data/train.csv', header=TRUE)
train <- data.matrix(train)
train.x <- train[,-1]
train.y <- train[,1]
train.x <- t(train.x/255)

data <- mx.symbol.Variable("data")
fc1 <- mx.symbol.FullyConnected(data, name="fc1", num_hidden=128)
act1 <- mx.symbol.Activation(fc1, name="relu1", act_type="relu")
fc2 <- mx.symbol.FullyConnected(act1, name="fc2", num_hidden=64)
act2 <- mx.symbol.Activation(fc2, name="relu2", act_type="relu")
fc3 <- mx.symbol.FullyConnected(act2, name="fc3", num_hidden=10)
softmax <- mx.symbol.SoftmaxOutput(fc3, name="sm")

devices <- mx.cpu()
mx.set.seed(0)
model <- mx.model.FeedForward.create(softmax, X=train.x, y=train.y,
                                     ctx=devices, num.round=10, array.batch.size=100,
                                     learning.rate=0.07, momentum=0.9,
                                     eval.metric=mx.metric.accuracy,
                                     initializer=mx.init.uniform(0.07),
                                     epoch.end.callback=mx.callback.log.train.metric(100))
```

Listing 1.2: Example MXNet code in R

1.3 PyTorch and TensorFlow

PyTorch and TensorFlow are the two major libraries nowadays. In the past, when TensorFlow was in version 1.x, they were vastly different. But since TensorFlow absorbed Keras as part of its library, these two libraries mostly work similarly.

PyTorch is backed by Facebook, and its syntax has been stable over the years. There are also a lot of existing models that we can borrow. The common way of defining a deep learning model in PyTorch is to create a class:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=(5,5), stride=1, padding=2)
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0)
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(16, 120, kernel_size=5, stride=1, padding=0)
        self.flatten = nn.Flatten()
        self.linear4 = nn.Linear(120, 84)
        self.linear5 = nn.Linear(84, 10)
        self.softmax = nn.LogSoftMax(dim=1)

    def forward(self, x):
        x = F.tanh(self.conv1(x))
        x = self.pool1(x)
        x = F.tanh(self.conv2(x))
        x = self.pool2(x)
        x = F.tanh(self.conv3(x))
        x = self.flatten(x)
        x = F.tanh(self.linear4(x))
        x = self.linear5(x)
        return self.softmax(x)

model = Model()
```

Listing 1.3: Example PyTorch code using sub-classing syntax

But there is also a sequential syntax to make the code more concise:

```python
import torch
import torch.nn as nn

model = nn.Sequential(
    # assume input 1x28x28
    nn.Conv2d(1, 6, kernel_size=(5,5), stride=1, padding=2),
    nn.Tanh(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
    nn.Tanh(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(16, 120, kernel_size=5, stride=1, padding=0),
    nn.Tanh(),
    nn.Flatten(),
    nn.Linear(120, 84),
    nn.Tanh(),
    nn.Linear(84, 10),
    nn.LogSoftmax(dim=1)
)
```

Listing 1.4: Example PyTorch code using sequential syntax

TensorFlow in version 2.x adopted Keras as part of its libraries. In the past, these two were separate projects. In TensorFlow 1.x, we need to build a computation graph, set up a session, and derive gradients from a session for the deep learning model. Hence it is a bit too verbose. Keras is designed as a library to hide all these low-level details.

The same network as above can be produced by TensorFlow's Keras syntax as follows:

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, AveragePooling2D, Flatten

model = Sequential([
    Conv2D(6, (5,5), input_shape=(28,28,1), padding="same", activation="tanh"),
    AveragePooling2D((2,2), strides=2),
    Conv2D(16, (5,5), activation="tanh"),
    AveragePooling2D((2,2), strides=2),
    Conv2D(120, (5,5), activation="tanh"),
    Flatten(),
    Dense(84, activation="tanh"),
    Dense(10, activation="softmax")
])
```

Listing 1.5: Example TensorFlow code using sequential syntax

One major difference between PyTorch and Keras syntax is in the training loop. In Keras, we just need to assign the loss function, the optimization algorithm, the dataset, and some other parameters to the model. Then we have a `fit()` function to do all the training work, as follows:

```python
...
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=100, batch_size=32)
```

Listing 1.6: Usign `fit()` function in TensorFlow

But in PyTorch, we need to write our own training loop code:

```python
# self-defined training loop function
def training_loop(model, optimizer, loss_fn, train_loader, val_loader=None, n_epochs=100):
    best_loss, best_epoch = np.inf, -1
    best_state = model.state_dict()
    for epoch in range(n_epochs):
        # Training
        model.train()
        train_loss = 0
        for data, target in train_loader:
            output = model(data)
            loss = loss_fn(output, target)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
        # Validation
        model.eval()
        status = (f"{str(datetime.datetime.now())} End of epoch {epoch}, "
                  f"training loss={train_loss/len(train_loader)}")
        if val_loader:
            val_loss = 0
            for data, target in val_loader:
                output = model(data)
                loss = loss_fn(output, target)
                val_loss += loss.item()
            status += f", validation loss={val_loss/len(val_loader)}"
        print(status)

optimizer = optim.Adam(model.parameters())
criterion = nn.NLLLoss()
training_loop(model, optimizer, criterion, train_loader, test_loader, n_epochs=100)
```

Listing 1.7: User-defined training loop function for PyTorch

This may not be an issue if you're experimenting with a new design of network in which you want to have more control over how the loss is calculated and how the optimizer updates the model weights. But otherwise, you will appreciate the simpler syntax from Keras.

Note that both PyTorch and TensorFlow are libraries with a Python interface. Therefore, it is possible to have an interface for other languages too. For example, there are Torch for R and TensorFlow for R.

Also, note that the libraries we mentioned above are full-featured libraries that include training and prediction. If you consider a production environment where you make use of a trained model, there could be a wider choice. TensorFlow has a "TensorFlow Lite" counterpart that allows a trained model to be run on a mobile device or the web. Intel also has an OpenVINO library that aims to optimize the performance in prediction.

1.4 Further Reading

Below are the links to the libraries we mentioned above:

libann.
    https://www.nongnu.org/libann/
OpenNN.
    https://www.opennn.net/

Chainer.
    https://chainer.org/
Caffe.
    https://caffe.berkeleyvision.org/
CNTK.
    https://docs.microsoft.com/en-us/cognitive-toolkit/
MXNet.
    https://mxnet.apache.org/versions/1.9.1/
Theano.
    https://github.com/Theano/Theano
PyTorch.
    https://pytorch.org/
TensorFlow.
    https://www.tensorflow.org/
Torch for R.
    https://torch.mlverse.org/
TensorFlow for R.
    https://tensorflow.rstudio.com/

1.5 Summary

In this chapter, you discovered various deep learning libraries and some of their characteristics. Specifically, you learned:

▷ What are the libraries available for C++ and Python

▷ How the Chainer library influenced the syntax in building a deep learning model

▷ The relationship between Keras and TensorFlow 2.x

▷ What are the differences between PyTorch and TensorFlow

In the next chapter, you will learn more about the library of choice in this book, TensorFlow.

2 Introduction to TensorFlow

TensorFlow is a Python library for fast numerical computing created and released by Google. It is a foundation library that can be used to create Deep Learning models directly or by using wrapper libraries that simplify the process built on top of TensorFlow. After completing this chapter, you will know:

- ▷ About the TensorFlow library for Python
- ▷ How to define, compile, and evaluate a simple symbolic expression in TensorFlow
- ▷ where to go to get more information on the library

Let's get started.

Overview

This chapter is in four parts; they are:

- ▷ What is TensorFlow?
- ▷ How to Install TensorFlow
- ▷ Your First Examples in TensorFlow
- ▷ More Deep Learning Models

2.1 What Is TensorFlow?

TensorFlow is an open-source library for fast numerical computing. It was created and is maintained by Google and was released under the Apache 2.0 open source license. The API is nominally for the Python programming language, although there is access to the underlying C++ API. Unlike other numerical libraries intended for use in Deep Learning like Theano, TensorFlow was designed for use both in research and development and in production systems, not least of which is RankBrain in Google search[1] and the fun DeepDream project[2]. It can run on single CPU systems and GPUs, as well as mobile devices and large-scale distributed systems of hundreds of machines.

[1]https://en.wikipedia.org/wiki/RankBrain
[2]https://en.wikipedia.org/wiki/DeepDream

2.2 How to Install TensorFlow

Installation of TensorFlow is straightforward if you already have a Python environment. If you need help setting up your workstation, see Appendix B.

TensorFlow works with Python 3.3+. You can follow the Download and Setup instructions on the TensorFlow website. Installation is probably simplest via PyPI, and specific instructions of the `pip` command to use for your Linux or Mac OS X platform are on the Download and Setup webpage. Usually, you just need to enter the following in your command line:

```
pip install tensorflow
```

An exception would be on the newer Mac with an Apple Silicon CPU. The package name for this specific architecture is `tensorflow-macos` instead:

```
pip install tensorflow-macos
```

There are also virtualenv and docker images that you can use if you prefer. To make use of the GPU, you need to have the CUDA Toolkit installed as well.

2.3 Your First Examples in TensorFlow

Computation in TensorFlow is described in terms of data flow and operations in the structure of a directed graph. The figure below is an example of how a mathematical expression can be represented.

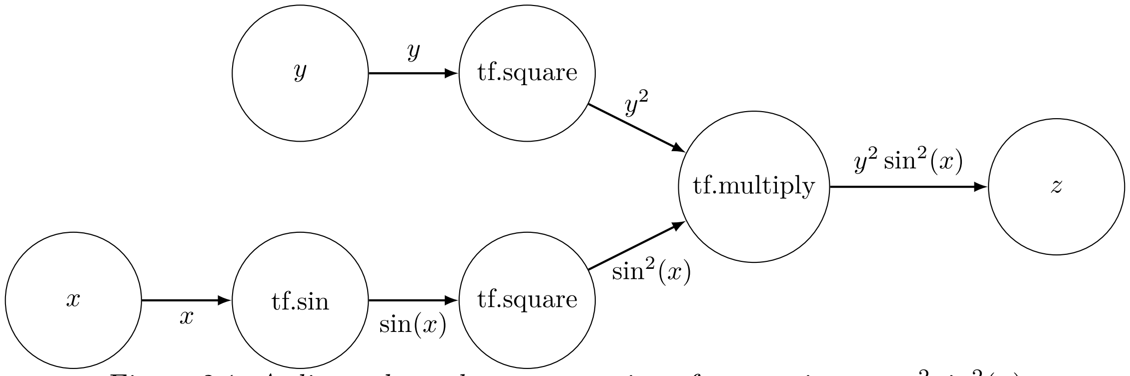


Figure 2.1: A directed graph representation of computing $z = y^2 \sin^2(x)$

There are some terms in describing a directed graph:

- ▷ **Nodes:** Nodes perform computation and have zero or more inputs and outputs. Data that moves between nodes are known as tensors, which are multi-dimensional arrays of real values.
- ▷ **Edges:** The graph defines the flow of data, branching, looping, and updates to state. Special edges can be used to synchronize behavior within the graph, for example, waiting for computation on a number of inputs to complete.

▷ **Operation:** An operation is a named abstract computation that can take input attributes and produce output attributes. For example, you could define an add or multiply operation.

We seldom use these terms when we write TensorFlow programs. But understanding them helps figuring out what TensorFlow does to our code.

Computation with TensorFlow

This first example is a modified version of an example on TensorFlow website. It shows how you can define values as *tensors* and execute an operation.

```python
import tensorflow as tf
a = tf.constant(10)
b = tf.constant(32)
print(a+b)
```

Listing 2.1: A simple TensorFlow program for addition

Running this example displays:

```
tf.Tensor(42, shape=(), dtype=int32)
```

Output 2.1: Result of TensorFlow addition

Linear Regression with TensorFlow

This next example comes from the introduction in the older version of TensorFlow tutorial.

This example shows how you can define variables (e.g., `W` and `b`) as well as variables resulting from computation (`y`). Below, there is automatic differentiation under the hood. When we use `mse_loss()` function to compute the differences between `y` and `y_data`, there is a graph created connecting the values produced by the function (`loss`) to the TensorFlow variables `W` and `b`. TensorFlow uses this graph to deduce how to update the variables inside the `minimize()` function.

```python
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# Try to find values for W and b that compute y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but Tensorflow will figure that out for us.)
W = tf.Variable(tf.random.normal([1]))
b = tf.Variable(tf.zeros([1]))

# A function to compute mean squared error between y_data and computed y
def mse_loss():
    y = W * x_data + b
    loss = tf.reduce_mean(tf.square(y - y_data))
    return loss

# Minimize the mean squared errors.
optimizer = tf.keras.optimizers.Adam()
for step in range(5000):
    optimizer.minimize(mse_loss, var_list=[W,b])
    if step % 500 == 0:
        print(step, W.numpy(), b.numpy())

# Learns best fit is W: [0.1], b: [0.3]
```

Listing 2.2: Linear regression in TensorFlow

Running this example prints the following output:

```
0 [-0.35913563] [0.001]
500 [-0.04056413] [0.3131764]
1000 [0.01548613] [0.3467598]
1500 [0.03492216] [0.3369852]
2000 [0.05408324] [0.32609695]
2500 [0.07121297] [0.316361]
3000 [0.08443557] [0.30884594]
3500 [0.09302785] [0.3039626]
4000 [0.09754606] [0.3013947]
4500 [0.09936733] [0.3003596]
```

Output 2.2: Progress of running linear regression using TensorFlow

You can learn more about the mechanics of TensorFlow in its Basic Usage guide.

2.4 More Deep Learning Models

Your TensorFlow installation comes with a number of Deep Learning models that you can use and experiment with directly. Firstly, you need to find out where TensorFlow was installed on your system. For example, you can use the following Python script on the command line:

```
python -c 'import os,tensorflow;print(os.path.dirname(tensorflow.__file__))'
```

Listing 2.3: Command to print TensorFlow install location

For example, this could be:

```
/usr/lib/python3.9/site-packages/tensorflow
```

Output 2.3: TensorFlow install location

Under the subdirectories of this location, you will find a number of deep learning models with detailed comments, such as:

▷ Multi-threaded word2vec mini-batched skip-gram model

▷ Multi-threaded word2vec unbatched skip-gram model

- ▷ CNN for the CIFAR-10 network
- ▷ Simple, end-to-end, LeNet-5-like convolutional MNIST model example
- ▷ Sequence-to-sequence model with an attention mechanism

Also, check the examples directory, which contains an example using the MNIST dataset.

There is also an excellent list of tutorials on the main TensorFlow website. They show how to use different neural network constructions and different datasets, and how to use the framework in various ways.

Finally, there is the TensorFlow playground where you can experiment with small neural network models right in your web browser.

2.5 TensorFlow Resources

Below are the resources mentioned in this chapter:

TensorFlow.
`https://www.tensorflow.org/`
Install TensorFlow 2.
`https://www.tensorflow.org/install`
TensorFlow project on GitHub.
`https://github.com/tensorflow/tensorflow`
TensorFlow tutorials.
`https://www.tensorflow.org/tutorials`
TensorFlow Basic Usage Guide.
`https://www.tensorflow.org/guide/basics`
TensorFlow Playground.
`https://playground.tensorflow.org/`

2.6 Summary

In this chapter, you discovered the TensorFlow Python library for deep learning.

You learned that it is a library for fast numerical computation, specifically designed for the types of operations required to develop and evaluate of large deep learning models.

In the next chapter, you will learn about the autograd function from TensorFlow and how it became the basis of deep learning training.

3 Using Autograd in TensorFlow to Solve a Regression Problem

We usually use TensorFlow to build a neural network. However, TensorFlow is not limited to this. Behind the scenes, TensorFlow is a tensor library with automatic differentiation capability. Hence you can easily use it to solve a numerical optimization problem with gradient descent, which is the algorithm to train a neural network. In this chapter, you will learn how TensorFlow's automatic differentiation engine, autograd, works. After finishing this chapter, you will learn:

- ▷ What is autograd in TensorFlow
- ▷ How to make use of autograd and an optimizer to solve an optimization problem

Let's get started.

Overview

This chapter is in three parts; they are:

- ▷ Autograd in TensorFlow
- ▷ Using Autograd for Polynomial Regression
- ▷ Using Autograd to Solve a Math Puzzle

3.1 Autograd in TensorFlow

In TensorFlow 2.x, you can define variables and constants as TensorFlow objects and build an expression with them. The expression is essentially a function of the variables. Hence you may derive its derivative function, i.e., the differentiation or the gradient. This feature is one of the many fundamental features in TensorFlow. The deep learning model will make use of this in the training loop.

It is easier to explain autograd with an example. In TensorFlow 2.x, you can create a constant matrix as follows:

```
import tensorflow as tf

x = tf.constant([1, 2, 3])
print(x)
print(x.shape)
print(x.dtype)
```

Listing 3.1: A constant matrix defined in TensorFlow

The above prints:

```
tf.Tensor([1 2 3], shape=(3,), dtype=int32)
(3,)
<dtype: 'int32'>
```

Output 3.1: Constant matrix created from Listing 3.1

This creates an integer vector (in the form of `Tensor` object). This vector can work like a NumPy vector in most cases. For example, you can do `x+x` or `2*x`, and the result is just what you would expect. TensorFlow comes with many functions for array manipulation that match NumPy, such as `tf.transpose` or `tf.concat`.

Creating variables in TensorFlow is just the same, for example:

```
import tensorflow as tf

x = tf.Variable([1, 2, 3])
print(x)
print(x.shape)
print(x.dtype)
```

Listing 3.2: A variable defined in TensorFlow

This will print:

```
<tf.Variable 'Variable:0' shape=(3,) dtype=int32, numpy=array([1, 2, 3], dtype=int32)>
(3,)
<dtype: 'int32'>
```

Output 3.2: Variable created from Listing 3.2

The operations (such as `x+x` and `2*x`) that you can apply to `Tensor` objects can also be applied to variables. The difference between variables and constants is that the former allows the value to change while the latter is immutable. This distinction is important when you run a *gradient tape* as follows:

```
import tensorflow as tf

x = tf.Variable(3.6)

with tf.GradientTape() as tape:
    y = x*x
```

```
dy = tape.gradient(y, x)
print(dy)
```

Listing 3.3: Getting gradient in TensorFlow

This prints:

```
tf.Tensor(7.2, shape=(), dtype=float32)
```

Output 3.3: Gradient as obtained from Listing 3.3

What it does is the following: This defined a variable `x` (with value 3.6) and then created a gradient tape. While the gradient tape is working, it computes `y=x*x` or $y = x^2$. The gradient tape monitored how the variables are manipulated. Afterward, you asked the gradient tape to find the derivative $\frac{dy}{dx}$. You know $y = x^2$ means $\frac{dy}{dx} = 2x$. Hence the output would give you a value of $3.6 \times 2 = 7.2$.

3.2 Using Autograd for Polynomial Regression

How this feature in TensorFlow helpful?

Let's consider a case where you have a polynomial in the form of $y = f(x)$, and you are given several (x, y) samples. How can you recover the polynomial $f(x)$? One way is to assume random coefficients for the polynomial and feed in the samples (x, y). If the polynomial is found, you should see the value of y matches $f(x)$. The closer they are, the closer your estimate is to the correct polynomial. This is indeed a numerical optimization problem such that you want to minimize the difference between y and $f(x)$. You can use gradient descent to solve it.

Let's consider an example. You can build a polynomial $f(x) = x^2 + 2x + 3$ in NumPy as follows:

```
import numpy as np

polynomial = np.poly1d([1, 2, 3])
print(polynomial)
```

Listing 3.4: Defining a polynomial in NumPy

This prints:

```
   2
1 x + 2 x + 3
```

Output 3.4: A polynomial created

You may use the polynomial as a function, such as:

```
print(polynomial(1.5))
```

Listing 3.5: Using a polynomial

And this prints `8.25`, for $(1.5)^2 + 2 \times (1.5) + 3 = 8.25$.

Now you can generate a number of samples from this function using NumPy:

```
N = 20   # number of samples

# Generate random samples between -10 to +10
X = np.random.uniform(-10, 10, size=(N,1))
Y = polynomial(X)
```

Listing 3.6: Making samples from a polynomial

In the above, both `X` and `Y` are NumPy arrays of the shape `(20,1)`, and they are related as $y = f(x)$ for the polynomial $f(x)$.

Now, assume you do not know what the polynomial is, except it is quadratic. And you want to recover the coefficients. Since a quadratic polynomial is in the form of $Ax^2 + Bx + C$, you have three unknowns to find. You can find them using the gradient descent algorithm you implement or an existing gradient descent optimizer. The following demonstrates how it works:

```
import tensorflow as tf

# Assume samples X and Y are prepared elsewhere

XX = np.hstack([X*X, X, np.ones_like(X)])

w = tf.Variable(tf.random.normal((3,1)))  # the 3 coefficients
x = tf.constant(XX, dtype=tf.float32)     # input sample
y = tf.constant(Y, dtype=tf.float32)      # output sample
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
print(w)

for _ in range(1000):
    with tf.GradientTape() as tape:
        y_pred = x @ w
        mse = tf.reduce_sum(tf.square(y - y_pred))
    grad = tape.gradient(mse, w)
    optimizer.apply_gradients([(grad, w)])

print(w)
```

Listing 3.7: Regression on samples to discover the polynomial

The `print` statement before the for loop gives three random number, such as

```
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[-2.1450958 ],
       [-1.1278448 ],
       [ 0.31241694]], dtype=float32)>
```

Output 3.5: Vector `w` is three random numbers

But the one after the for loop gives you the coefficients very close to that in the polynomial:

```
<tf.Variable 'Variable:0' shape=(3, 1) dtype=float32, numpy=
array([[1.0000628],
       [2.0002015],
       [2.996219 ]], dtype=float32)>
```

Output 3.6: Vector `w` as the result of regression

What the above code does is the following: First, it creates a variable vector `w` of 3 values, namely the coefficients A, B, C. Then you create an array of shape $(N, 3)$, in which N is the number of samples in the array `X`. This array has 3 columns, which are the values of x^2, x, and 1, respectively. Such an array is built from the vector `X` using the `np.hstack()` function. Similarly, we build the TensorFlow constant `y` from the NumPy array `Y`. Afterwards, you use a for-loop to run gradient descent in 1,000 iterations. In each iteration, you compute $x \times w$ in matrix form to find $Ax^2 + Bx + C$ and assign it to the variable `y_pred`. Then, compare `y` and `y_pred` and find the mean square error. Next, derive the gradient, i.e., the rate of change of the mean square error with respect to the coefficients `w`. And based on this gradient, you use gradient descent to update `w`.

In essence, the above code is to find the coefficients `w` that minimizes the mean square error. Putting everything together, the following is the complete code:

```python
import numpy as np
import tensorflow as tf

N = 20   # number of samples

# Generate random samples between -10 to +10
polynomial = np.poly1d([1, 2, 3])
X = np.random.uniform(-10, 10, size=(N,1))
Y = polynomial(X)

# Prepare input as an array of shape (N,3)
XX = np.hstack([X*X, X, np.ones_like(X)])

# Prepare TensorFlow objects
w = tf.Variable(tf.random.normal((3,1)))  # the 3 coefficients
x = tf.constant(XX, dtype=tf.float32)     # input sample
y = tf.constant(Y, dtype=tf.float32)      # output sample
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
print(w)

# Run optimizer
for _ in range(1000):
    with tf.GradientTape() as tape:
        y_pred = x @ w
        mse = tf.reduce_sum(tf.square(y - y_pred))
    grad = tape.gradient(mse, w)
    optimizer.apply_gradients([(grad, w)])

print(w)
```

Listing 3.8: Regression to discover a polynomial using TensorFlow

3.3 Using Autograd to Solve a Math Puzzle

In the above, 20 samples were used, which is more than enough to fit a quadratic equation. You may use gradient descent to solve some math puzzles as well. For example, the following problem:

$$\begin{array}{ccccc} A & + & B & = & 8 \\ + & & + & & \\ C & - & D & = & 6 \\ = & & = & & \\ 13 & & 8 & & \end{array}$$

In other words, to find the values of A, B, C, D such that:

$$A + B = 8$$

$$C - D = 6$$

$$A + C = 13$$

$$B + D = 8$$

This can also be solved using autograd, as follows:

```python
import tensorflow as tf
import random

A = tf.Variable(random.random())
B = tf.Variable(random.random())
C = tf.Variable(random.random())
D = tf.Variable(random.random())

# Gradient descent loop
EPOCHS = 1000
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.1)
for _ in range(EPOCHS):
    with tf.GradientTape() as tape:
        y1 = A + B - 8
        y2 = C - D - 6
        y3 = A + C - 13
        y4 = B + D - 8
        sqerr = y1*y1 + y2*y2 + y3*y3 + y4*y4
    gradA, gradB, gradC, gradD = tape.gradient(sqerr, [A, B, C, D])
    optimizer.apply_gradients([(gradA, A), (gradB, B), (gradC, C), (gradD, D)])

print(A)
print(B)
print(C)
print(D)
```

Listing 3.9: Solving a math puzzle using TensorFlow

There can be multiple solutions to this problem. One solution is the following:

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=3.500003>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=4.5006905>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=9.499581>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=3.540172>
```

Output 3.7: Solution as found by Listing 3.9

Which means $A = 3.5$, $B = 4.5$, $C = 9.5$, and $D = 3.5$. You can verify this solution fits the problem.

The above code defines the four unknown as variables with random initial values. Then you compute the result of the four equations and compare it to the expected answer. You then sum up the squared error and ask TensorFlow to minimize it. The minimum possible square error is zero, attained when our solution exactly fits the problem.

Note the way the gradient tape is asked to produce the gradient: You ask the gradient of `sqerr` respective to `A`, `B`, `C`, and `D`. Hence four gradients are found. You then apply each gradient to the respective variables in each iteration. Rather than looking for the gradient in four different calls to `tape.gradient()`, this is required in TensorFlow because the gradient of `sqerr` can only be recalled once by default.

3.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Articles

Introduction to gradients and automatic differentiation. TensorFlow.
https://www.tensorflow.org/guide/autodiff

Advanced automatic differentiation. TensorFlow.
https://www.tensorflow.org/guide/advanced_autodiff

3.5 Summary

In this chapter, we demonstrated how TensorFlow's automatic differentiation works. This is the building block for carrying out deep learning training. Specifically, you learned:

▷ What is automatic differentiation in TensorFlow

▷ How you can use gradient tape to carry out automatic differentiation

▷ How you can use automatic differentiation to solve a optimization problem

In the next chapter, you will learn about the higher-level library for deep learning, Keras.

4 Introduction to Keras

TensorFlow is a very powerful library, but can be difficult to use directly for creating deep learning models. In this chapter, you will discover the Keras Python library that provides a clean and convenient way to create a range of deep learning models on top of TensorFlow. After completing this chapter, you will know:

- ▷ About the Keras Python library for deep learning.
- ▷ The standard idiom for creating models with Keras.

Let's get started.

Overview

This chapter is in three parts; they are:

- ▷ What is Keras?
- ▷ How to Install Keras
- ▷ Using Autograd to Solve a Math Puzzle

4.1 What is Keras?

Keras is a minimalist Python library for deep learning that can run on top of TensorFlow. It was developed to make implementing deep learning models as fast and easy as possible for research and development. It runs on Python and can seamlessly execute on GPUs and CPUs through TensorFlow. It is released under the permissive MIT license.

Keras was developed and maintained by François Chollet, a Google engineer, using four guiding principles:

- ▷ **Modularity**: A model can be understood as a sequence or a graph alone. All the concerns of a deep learning model are discrete components that can be combined in arbitrary ways.
- ▷ **Minimalism**: The library provides just enough to achieve an outcome, no frills and maximizing readability.

- ▷ **Extensibility**: New components are intentionally easy to add and use within the framework, intended for researchers to trial and explore new ideas.
- ▷ **Python**: No separate model files with custom file formats. Everything is native Python.

4.2 How to Install Keras

Keras is relatively straightforward to install if you already have a working Python environment. If you need help setting up your workstation, see Appendix B. In the past, Keras was a standalone library. Nowadays, it is part of TensorFlow. Therefore you just need to install TensorFlow using `pip`, as follows:

```
pip install tensorflow
```

For backward compatibility reason, Keras can also be installed as follows:

```
pip install keras
```

And functionally you installed the same library.

At the time of writing, the most recent version of TensorFlow is version 2.9.2. You can check your version of TensorFlow on the command line using the following snippet:

```
python -c "import tensorflow; print(tensorflow.__version__)"
```

Listing 4.1: Checking the version of TensorFlow installed

Running the above script you will see:

```
2.9.2
```

Output 4.1: TensorFlow version

You can upgrade your installation of TensorFlow/Keras using the same method:

```
pip install --upgrade tensorflow
```

4.3 Build Deep Learning Models with Keras

With TensorFlow, you can build a neural network by creating variables for the weights and define how they interact with the training data as input to produce outputs. You already saw how to use TensorFlow this way in Chapter 3. However, this is too tedious and inefficient. Using Keras helps you to think in terms of layers instead of individual weights and parameters.

Keras centers on the concept of a model. The main model type is a called a `Sequential` which is a linear stack of layers. You create a `Sequential` and add layers to it in the order that the computation should be performed. Once defined, you compile the model to make use of the underlying framework to optimize computations to be performed. In this you can specify the loss function and the optimizer to be used.

Once compiled, the model must be fit to data. This can be done one batch of data at a time or by firing off the entire model training regime. This is where all the compute happens. Once trained, you can use your model to make predictions on new data. We can summarize the construction of deep learning models in Keras as follows:

1. **Define your model**. Create a `Sequential` model and add configured layers.
2. **Compile your model**. Specify loss function and optimizers and call the `compile()` function on the model.
3. **Fit your model**. Train the model on a sample of data by calling the `fit()` function on the model.
4. **Make predictions**. Use the model to generate predictions on new data by calling functions such as `evaluate()` or `predict()` on the model.

4.4 Keras Resources

The list below provides some additional resources that you can use to learn more about Keras.

tf.keras module. TensorFlow.
https://www.tensorflow.org/api_docs/python/tf/keras

Keras official homepage.
https://keras.io

Keras project on GitHub.
https://github.com/keras-team/keras

4.5 Summary

In this chapter, you discovered the Keras Python library for deep learning research and development. You learned:

- ▷ Keras is a high-level library for TensorFlow, abstracting their capabilities and hiding their complexity.
- ▷ Keras is designed for minimalism and modularity, allowing you to very quickly define deep learning models.
- ▷ Keras deep learning models can be developed using an idiom of defining, compiling and fitting models that can then be evaluated or used to make predictions.

Begining with next chapter, you will see how Keras can be used to build various deep learning models.

II
Multilayer Perceptrons

Understanding Multilayer Perceptrons

5

Artificial neural networks form a fascinating area of study, although they can be intimidating when you are just getting started. There are a lot of specialized terms in use to describe the data structures and algorithms used in the field. In this chapter, you will get a crash course in the terminology and processes used in the field of multilayer perceptron artificial neural networks. After completing this chapter, you will know:

- ▷ The building blocks of neural networks, including neurons, weights, and activation functions
- ▷ How the building blocks are used in layers to create networks
- ▷ How networks are trained from example data

Let's get started.

Overview

This chapter is in four parts; they are:

- ▷ Multilayer Perceptrons
- ▷ Neurons: Weights and Activations
- ▷ Networks of Neurons
- ▷ Training Networks

5.1 Multilayer Perceptrons

The field of artificial neural networks is often just called *neural networks* or *multilayer perceptrons*, as multilayer perceptron is perhaps the most useful type of neural network. A perceptron is a single neuron model that was a precursor to larger neural networks. It is a field of study that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks we see in machine learning. The goal is not to create realistic models of the brain but instead to develop robust algorithms and data structures that we can use to model difficult problems.

The power of neural networks come from their ability to learn the correlations in your training data and how best to relate it to the output variable you want to predict. In this sense, neural networks learn mapping. Mathematically, they are capable of learning any mapping function and have been proven to be a universal approximation algorithm. The predictive capability of neural networks comes from the hierarchical or multilayered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features, for example, from lines to collections of lines to shapes.

5.2 Neurons: Weights and Activations

The building blocks for neural networks are artificial neurons. These are simple computational units that have weighted input signals and produce an output signal using an activation function.

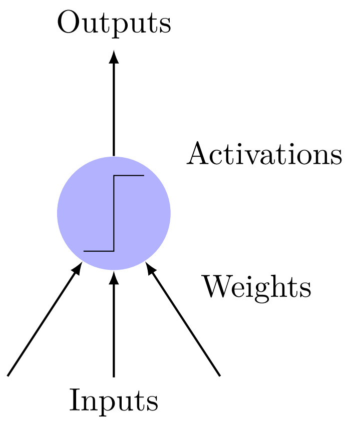


Figure 5.1: Model of a simple neuron

Neuron Weights

You may be familiar with linear regression, where the weights on the inputs are very much like the coefficients used in a regression equation. Like linear regression, each neuron also has a bias which can be thought of as an input that always has the value 1.0, and it, too, must be weighted. For example, a neuron may have two inputs, which requires three weights — one for each input and one for the bias.

Weights are often initialized to small random values, such as values from 0 to 0.3, although more complex initialization schemes can be used. Like linear regression, larger weights indicate increased complexity and fragility of the model. Keeping weights in the network small is desirable, and regularization techniques can be used.

Activation

The weighted inputs are summed and passed through an activation function, sometimes called a transfer function. An activation function is a simple mapping of summed weighted input to the output of the neuron. It is called an activation function because it governs the threshold at which the neuron is activated and the strength of the output signal. Historically, simple step activation functions were used. When the summed input was above a threshold of 0.5, for example, then the neuron would output a value of 1.0; otherwise it would output a 0.0.

Traditionally, nonlinear activation functions are used. This allows the network to combine the inputs in more complex ways and, in turn, provide a richer capability in the functions they can model. Nonlinear functions like the logistic function, also called the sigmoid function, were used to output a value between 0 and 1 with an S-shaped distribution. The hyperbolic tangent function, also called tanh, were used to output the same distribution over the range -1 to $+1$. More recently, the rectifier activation function has been shown to provide better results.

5.3 Networks of Neurons

Neurons are arranged into networks of neurons, the figure below is an example. A row of neurons is called a layer, and one network can have multiple layers. The architecture of the neurons in the network is often called the network topology.

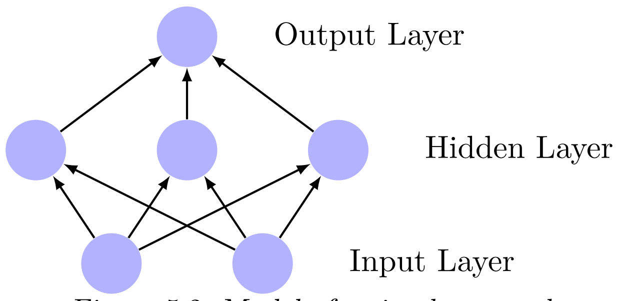


Figure 5.2: Model of a simple network

Input or Visible Layers

The bottom layer that takes input from your dataset is called the visible layer because it is the exposed part of the network. Often a neural network is drawn with a visible layer with one neuron per input value or column in your dataset. These are not actual neurons as described above, but simply drawn as nodes to represent input values that will be passed to the next layer.

Hidden Layers

Layers after the input layer are called hidden layers because they are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value. Given increases in computing power and efficient libraries, very deep neural networks can be constructed. Deep learning can refer to having many hidden layers in your neural network. They are deep because they would have been unimaginably slow to train historically but may take seconds or minutes to train using modern techniques and hardware.

Output Layer

The final hidden layer is called the output layer, and it is responsible for outputting a value or vector of values that correspond to the format required for the problem. The choice of activation function in the output layer is strongly constrained by the type of problem that you are modeling. For example:

- ▷ A regression problem may have a single output neuron, and the neuron may have no activation function.
- ▷ A binary classification problem may have a single output neuron and use a sigmoid activation function to output a value between 0 and 1 to represent the probability of the primary class as predicted. This can be turned into a crisp class value by using a threshold of 0.5 and snap values less than the threshold to 0, otherwise to 1.
- ▷ A multiclass classification problem may have multiple neurons in the output layer, one for each class (e.g., three neurons for the three classes in the famous iris flowers classification problem). In this case, a softmax activation function may be used to output a probability of the network predicting each of the class values. Selecting the output with the highest probability can be used to produce a crisp classification value.

5.4 Training Networks

Once configured, the neural network needs to be trained on your dataset.

Data Preparation

You must first prepare your data for training on a neural network. Data must be numerical, for example, real values. If you have categorical data, such as a *sex* attribute with the values “male” and “female”, you can convert it to a real-valued representation called one-hot encoding. This is where one new column is added for each class value (two columns in the case of sex of male and female), and a 0 or 1 is added for each row depending on the class value for that row.

This same one-hot encoding can be used on the output variable in classification problems with more than one class. This would create a binary vector from a single column that would be easy to directly compare to the output of the neuron in the network’s output layer. That, as described above, would output one value for each class. Neural networks require the input to be scaled in a consistent way. You can rescale it to the range between 0 and 1, this is called normalization. Another popular technique is to standardize it so that the distribution of each column has a mean of zero and a standard deviation of 1. Scaling also applies to image pixel data. Data such as words can be converted to integers, such as the frequency rank of the word in the dataset and other encoding techniques.

Stochastic Gradient Descent

The classical and still preferred training algorithm for neural networks is called stochastic gradient descent. This is where one row of data is exposed to the network at a time as input. The network processes the input upward, activating neurons as it goes to finally produce an output value. This is called a forward pass on the network. It is the type of pass that is also used after the network is trained in order to make predictions on new data.

The output of the network is compared to the expected output, and an error is calculated. This error is then propagated back through the network, one layer at a time, and the weights are updated according to the amount they contributed to the error. This clever bit of math is called the backpropagation algorithm. The process is repeated for all of the examples in your training data. One round of updating the network for the entire training dataset is called an epoch. A network may be trained for tens, hundreds, or many thousands of epochs.

Weight Updates

The weights in the network can be updated from the errors calculated for each training example, and this is called *online learning*. It can result in fast but also chaotic changes to the network. Alternatively, the errors can be saved across all the training examples, and the network can be updated at the end. This is called *batch learning* and is often more stable.

Because datasets are so large and because of computational efficiencies, the size of the batch, the number of examples the network is shown before an update, is often reduced to a small number, such as tens or hundreds of examples. The amount that weights are updated is controlled by a configuration parameter called the *learning rate*. It is also called the *step size* and controls the step or change made to network weights for a given error. Often small learning rates are used, such as 0.1 or 0.01 or smaller. The update equation can be complemented with additional configuration terms that you can set.

- **Momentum** is a term that carry over the weight update of the previous step to the current, so the weight update will continue in the same direction even when there is less error being calculated.
- **Learning Rate Decay** is used to decrease the learning rate over epochs to allow the network to make large changes to the weights at the beginning and smaller fine-tuning changes later in the training schedule.

Prediction

Once a neural network has been trained, it can be used to make predictions. You can make predictions on test or validation data in order to estimate the skill of the model on unseen data. You can also deploy it operationally and use it to make predictions continuously. The network topology and the final set of weights are all you need to save from the model. Predictions are made by providing the input to the network and performing a forward-pass, allowing it to generate an output you can use as a prediction.

5.5 More Resources

There are decades of papers and books on the topic of artificial neural networks. If you are new to the field, check out the following resources as further reading:

Books

Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1996.
https://www.amazon.com/dp/0198538642

Russell D. Reed and Robert J. Marks II. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, 1999.
https://www.amazon.com/dp/0262527014

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
https://www.amazon.com/dp/0262035618
(Online version at http://www.deeplearningbook.org).

5.6 Summary

In this chapter, you discovered artificial neural networks for machine learning. You learned:

- ▷ How neural networks are not models of the brain but are instead computational models for solving complex machine learning problems.
- ▷ That neural networks are comprised of neurons that have weights and activation functions.
- ▷ The networks are organized into layers of neurons and are trained using stochastic gradient descent.
- ▷ That it is a good idea to prepare your data before training a neural network model.

In the next chapter you will learn the structure and the building blocks of a multilayer perceptron model in Keras.

6 Building Multilayer Perceptron Models in Keras

The Keras Python library for deep learning focuses on the creating models as a sequence of layers. In this chapter, you will discover the simple components you can use to create neural networks and simple deep learning models using Keras from TensorFlow.

Let's get started.

Overview

This chapter is in seven parts; they are:

▷ Neural Network Models in Keras

▷ Model Input

▷ Model Layers

▷ Model Compilation

▷ Model Training

▷ Model Prediction

▷ Summarize the model

6.1 Neural Network Models in Keras

The focus of the Keras library is a model. The simplest model is defined in the `Sequential` class, which is a linear stack of Layers. You can create a `Sequential` model and define all the layers in the constructor; for example:

```
from tensorflow.keras.models import Sequential
model = Sequential(...)
```

Listing 6.1: Setting up a `Sequential` model in Keras

A more useful idiom is to create a Sequential model and add your layers in the order of the computation you wish to perform; for example:

```
from tensorflow.keras.models import Sequential
model = Sequential()
model.add(...)
model.add(...)
model.add(...)
```

Listing 6.2: Adding layers into a `Sequential` model

6.2 Model Inputs

The first layer in your model must specify the shape of the input. This is the number of input attributes defined by the `input_shape` argument. This argument expects a tuple. For example, you can define input in terms of 8 inputs for a `Dense` type layer as follows:

```
from tensorflow.keras.layers import Dense
layer = Dense(16, input_shape=(8,))
```

Listing 6.3: Defining a fully-connected layer

6.3 Model Layers

Layers of different types have a few properties in common, specifically their method of weight initialization and activation functions.

Weight Initialization

The type of initialization used for a layer is specified in the `kernel_initializer` argument. Some common types of layer initialization include:

- ▷ `random_uniform`: Weights are initialized to small uniformly random values between -0.05 and $+0.05$.
- ▷ `random_normal`: Weights are initialized to small Gaussian random values (zero mean and standard deviation of 0.05).
- ▷ `zero`: All weights are set to zero values.

For example, you can create a layer initialized with random uniform weights:

```
from tensorflow.keras.layers import Dense
layer = Dense(16, input_shape=(8,), kernel_initializer="random_uniform")
```

Listing 6.4: Defining a fully-connected layer with random unform weight

You can see a full list of the initialization techniques supported on the *Layer Weight Initializers* page.

Activation Function

Keras supports a range of standard neuron activation functions, such as softmax, rectified linear (ReLU), tanh, and sigmoid. You typically specify the type of activation function used by a layer in the `activation` argument, which takes a string value. You can see a full list of activation functions supported by Keras on the *Layer activation functions* page.

A layer usually has no activation by default, or effectively using a linear activation of $f(x) = x$. We need to specify an activation function if it is not desirable, as follows:

```
from tensorflow.keras.layers import Dense
layer = Dense(16, input_shape=(8,),
              kernel_initializer="random_uniform", activation="relu")
```

Listing 6.5: Defining a fully-connected layer with random unform weight and ReLU activation

Interestingly, you can also create an `Activation` object and add it directly to your model after your layer to apply that activation to the output of the layer.

Layer Types

There are a large number of core layer types for standard neural networks. Some common and useful layer types you can choose from are:

- ▷ `Dense`: Fully connected layer and the most common type of layer used on multilayer perceptron models. This is the one you saw above
- ▷ `Dropout`: Apply dropout to the model, setting a fraction of inputs to zero in an effort to reduce overfitting
- ▷ `Concatenate`: Combine the outputs from multiple layers as input to a single layer

You can learn about the full list of core Keras layers on the *Keras Layers API* page.

6.4 Model Compilation

Once you have defined your model, it needs to be compiled. This creates the efficient structures used by TensorFlow in order to efficiently execute your model during training. Specifically, TensorFlow converts your model into a graph so the training can be carried out efficiently.

You compile your model using the `compile()` function, and it accepts three important attributes:

1. Model optimizer
2. Loss function
3. Metrics

```
model.compile(optimizer=..., loss=..., metrics=...)
```

Listing 6.6: Syntax of compiling a Keras model

Model Optimizers

The optimizer is the search technique used to update weights in your model. You can create an optimizer object and pass it to the compile function via the optimizer argument. This allows you to configure the optimization procedure with its own arguments, such as learning rate. For example:

```
from tensorflow.keras.optimizers import SGD
sgd = SGD(...)
model.compile(optimizer=sgd)
```

Listing 6.7: Assigning an optimizer to a Keras model

You can also use the default parameters of the optimizer by specifying the name of the optimizer to the optimizer argument. For example:

```
model.compile(optimizer='sgd')
```

Listing 6.8: Using SGD optimizer via its name

Some popular gradient descent optimizers you might want to choose from include:

▷ `sgd`: stochastic gradient descent, with support for momentum

▷ `rmsprop`: adaptive learning rate optimization method proposed by Geoff Hinton

▷ `adam`: Adaptive Moment Estimation (Adam) that also uses adaptive learning rates

You can learn about all of the optimizers supported by Keras on the *Optimizers* page.

You can learn more about different gradient descent methods in the Gradient descent optimization algorithms section of Sebastian Ruder's post *An overview of gradient descent optimization algorithms*.

6.4.1 Model Loss Functions

The loss function, also called the objective function, is the evaluation of the model used by the optimizer to navigate the weight space. You can specify the name of the loss function to use in the compile function by the loss argument. Some common examples include:

▷ `mean_squared_error`: for mean squared error

▷ `binary_crossentropy`: for binary logarithmic loss (logloss)

▷ `categorical_crossentropy`: for multiclass logarithmic loss (logloss)

You can learn more about the loss functions supported by Keras on the *Losses* page.

Model Metrics

Metrics are evaluated by the model during training. Usually one metric is supported at the moment, and that is accuracy.

6.5 Model Training

The model is trained on NumPy arrays using the `fit()` function; for example:

```
model.fit(X, y, epochs=..., batch_size=...)
```

Training both specifies the number of epochs to train on and the batch size.

▷ Epochs (`epochs`) refer to the number of times the model is exposed to the training dataset.

▷ Batch Size (`batch_size`) is the number of training instances shown to the model before a weight update is performed.

The fit function also allows for some basic evaluation of the model during training. You can set the `validation_split` value to hold back a fraction of the training dataset for validation to be evaluated in each epoch or provide a `validation_data` tuple of `(X, y)` data to evaluate.

Fitting the model returns a history object with details and metrics calculated for the model in each epoch. This can be used for graphing model performance.

6.6 Model Prediction

Once you have trained your model, you can use it to make predictions on test data or new data. You can either call the `evaluate()` or the `predict()` function from the model:

▷ `model.evaluate(X, y)`: To calculate the loss values for the input data

▷ `model.predict(X)`: To generate network output for the input data

For example, if you provided a batch of data `X` and the expected output `y`, you can use `evaluate()` to calculate the loss metric (the one you defined with `compile()` before). But for a batch of new data `X`, you can obtain the network output with `predict()`. It may not be the output you want but it will be the output of your network. For example, a classification problem will probably output a softmax vector for each sample. You will need to use `numpy.argmax()` to convert the softmax vector into class labels.

6.7 Summarize the Model

Once you are happy with your model, you can finalize it. You may wish to output a summary of your model. For example, you can display a summary of a model to your terminal screen by calling the summary function:

```
model.summary()
```

Listing 6.9: Printing a Keras model

You can also retrieve a summary of the model configuration using the `get_config()` function:

```
config = model.get_config()
```

Listing 6.10: Retrieve a model configuration as a Python dict

Finally, you can create an image of your model structure directly:

```
from tensorflow.keras.utils import plot_model
plot_model(model, to_file='model.png',
           show_shapes=True, show_dtype=True, show_layer_names=True,
           expand_nested=True, show_layer_activations=True)
```

Listing 6.11: Creating an image file representing a model structure

6.8 Resources

You can learn more about how to create simple neural network and deep learning models in Keras using the following resources:

Articles

Layer Weight Initializers. Keras API Reference.
https://keras.io/api/layers/initializers/

Layer activation functions. Keras API Reference.
https://keras.io/api/layers/activations/

Keras Layers API. Keras API Reference.
https://keras.io/api/layers/

Optimizers. Keras API Reference.
https://keras.io/api/optimizers/

Sebastian Ruder. *An overview of gradient descent optimization algorithms*.
https://ruder.io/optimizing-gradient-descent/

Losses. Keras API Reference.
https://keras.io/api/losses/

The Sequential class. Keras API Reference.
https://keras.io/api/models/sequential/

The Sequential Model. Keras Developer Guides.
https://keras.io/guides/sequential_model/

The Models API. Keras API Reference.
https://keras.io/api/models/

6.9 Summary

In this chapter, you discovered the Keras API that you can use to create artificial neural networks and deep learning models. Specifically, you learned about the life cycle of a Keras model, including:

▷ Constructing a model

▷ Creating and adding layers including weight initialization and activation

- Compiling models including optimization method, loss function and metrics
- Fitting models including epochs and batch size
- Model predictions
- Summarizing the model

You now know the basics of neural network models. In the next chapter you will develop your very first multilayer perceptron model in Keras.

7 Develop Your First Neural Network with Keras

Keras is a powerful and easy-to-use free open source Python library for developing and evaluating *deep learning models*. It is part of the TensorFlow library and allows you to define and train neural network models in just a few lines of code. In this chapter, you will discover how to create your first deep learning neural network model in Python using Keras. After completing this chapter you will know:

- How to load a CSV dataset ready for use with Keras.
- How to define and compile a Multilayer Perceptron model in Keras.
- How to evaluate a Keras model on a validation dataset.

Let's get started.

Overview

There is not a lot of code required, but we will go over it slowly so that you will know how to create your own models in the future. The steps you will learn in this chapter are as follows:

- Load Data
- Define Keras Model
- Compile Keras Model
- Fit Keras Model
- Evaluate Keras Model
- Tie It All Together
- Make Predictions

7.1 Load Data

The first step is to define the functions and classes you intend to use in this chapter. You will use the NumPy library to load your dataset and two classes from the Keras library to define your model.

The imports required are listed below.

```
# first neural network with keras
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
...
```

Listing 7.1: Import statements used in this chapter

You can now load our dataset.

In this Keras tutorial, you will use the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years.

It is a binary classification problem (onset of diabetes as 1 or not as 0). All of the input variables that describe each patient are numerical. This makes it easy to use directly with neural networks that expect numerical input and output values and is an ideal choice for our first neural network in Keras.

The dataset is available from the bundle of sample code provided with this book. You can also download it here:

▷ Dataset CSV File (`https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv`)

▷ Dataset Details (`https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.names`)

Download the dataset and place it in your local working directory, the same location as your Python file. Save it with the filename `pima-indians-diabetes.csv`. Take a look inside the file; you should see rows of data like the following:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
...
```

Output 7.1: Samples of the Pima Indians dataset

You can now load the file as a matrix of numbers using the NumPy function `loadtxt()`. There are eight input variables and one output variable (the last column). You will be learning a model to map rows of input variables (X) to an output variable (y), which is often summarized as $y = f(X)$. The variables can be summarized as follows:

Input Variables (X):

1. Number of times pregnant
2. Plasma glucose concentration at 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)

5. 2-hour serum insulin (μIU/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)

Output Variables (y):

- ▷ Class variable (0 or 1)

Once the CSV file is loaded into memory, you can split the columns of data into input and output variables.

The data will be stored in a 2D array where the first dimension is rows and the second dimension is columns, e.g., (rows, columns). You can split the array into two arrays by selecting subsets of columns using the standard NumPy slice operator "`:`". You can select the first eight columns from index 0 to index 7 via the slice `0:8`. You can then select the output column (the 9th variable) via index 8.

```
...
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
...
```

Listing 7.2: Load the dataset using NumPy

You are now ready to define your neural network model.

7.2 Define Keras Model

Models in Keras are defined as a sequence of layers. You create a `Sequential` model and add layers one at a time until you are happy with our network architecture. The first thing to get right is to ensure the input layer has the correct number of input features. This can be specified when creating the first layer with the `input_shape` argument and setting it to `(8,)` for presenting the eight input variables as a vector.

How do you know the number of layers and their types? This is a tricky question. There are heuristics that you can use, and often the best network structure is found through a process of trial and error experimentation. Generally, you need a network large enough to capture the structure of the problem. In this example, let's use a fully-connected network structure with three layers.

Fully connected layers are defined using the `Dense` class. You can specify the number of neurons or nodes in the layer as the first argument and the activation function using the `activation` argument. Also, you will use the *rectified linear unit* activation function, referred to as ReLU, on the first two layers and the sigmoid function in the output layer.

It used to be the case that sigmoid and tanh activation functions were preferred for all layers. These days, better performance is achieved using the ReLU activation function. Using a sigmoid on the output layer ensures your network output is between 0 and 1, and is easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5. You can piece it all together by adding each layer:

- ▷ The model expects rows of data with 8 variables (the `input_shape=(8,)` argument).
- ▷ The first hidden layer has 12 nodes and uses the relu activation function.
- ▷ The second hidden layer has 8 nodes and uses the relu activation function.
- ▷ The output layer has one node and uses the sigmoid activation function.

```
...
# define the keras model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
...
```

Listing 7.3: Define the neural network model in Keras

You are free to change the design and see if you get a better or worse result than the subsequent part of this chapter. The figure below provides a depiction of the network structure:

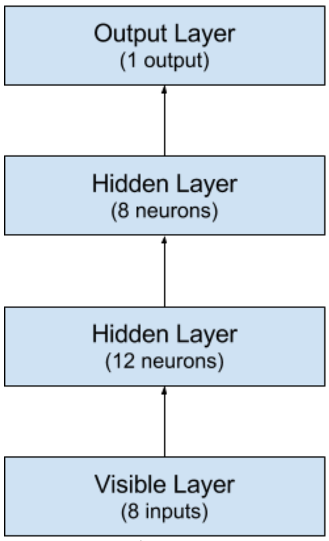


Figure 7.1: Model of a simple neural network

Note: The most confusing thing here is that the shape of the input to the model is defined as an argument on the first hidden layer. This means that the line of code that adds the first `Dense` layer is doing two things, defining the input or visible layer and the first hidden layer.

7.3 Compile Keras Model

Now that the model is defined, you can compile it. Compiling the model uses TensorFlow as an efficient numerical libraries (also called the backend). The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware, such as CPU, GPU, or even distributed. When compiling, you must specify some additional properties required when training the network. Remember training a network means finding the best set of weights to map inputs to outputs in your dataset.

You must specify the loss function to use to evaluate a set of weights, the optimizer is used to search through different weights for the network, and any optional metrics you want to collect and report during training. In this case, use cross-entropy as the `loss` argument. This loss is for a binary classification problems and is defined in Keras as "`binary_crossentropy`". You will define the `optimizer` as the efficient stochastic gradient descent algorithm "`adam`". This is a popular version of gradient descent because it automatically tunes itself and gives good results in a wide range of problems. Finally, because it is a classification problem, you will collect and report the classification accuracy defined via the `metrics` argument.

```
...
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
...
```

Listing 7.4: Compile the neural network model

7.4 Fit Keras Model

You have defined your model and compiled it to get ready for efficient computation. Now it is time to execute the model on some data. You can train or fit your model on your loaded data by calling the `fit()` function on the model.

Training occurs over epochs, and each epoch is split into batches.

- ▷ **Epoch**: One pass through all of the rows in the training dataset
- ▷ **Batch**: One or more samples considered by the model within an epoch before weights are updated

One epoch comprises of one or more batches, based on the chosen batch size, and the model is fit for many epochs. The training process will run for a fixed number of epochs (iterations) through the entire dataset that you must specify using the `epochs` argument. You must also set the number of dataset rows that are considered before the model weights are updated within each epoch, called the batch size, and set using the `batch_size` argument. This problem will run for a small number of epochs (150) and use a relatively small batch size of 10. These configurations can be chosen experimentally by trial and error. You want to train the model enough so that it learns a good (or good enough) mapping of rows of input data to the output classification. The model will always have some error, but the amount of error will level out after some point for a given model configuration. This is called *model convergence*.

```
...
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
...
```

Listing 7.5: Fit the neural network model to the dataset

This is where the work happens on your CPU or GPU.

7.5 Evaluate Keras Model

You have trained our neural network on the entire dataset, and you can evaluate the performance of the network on the same dataset. This will only give you an idea of how well you have modeled the dataset (e.g., train accuracy), but no idea of how well the algorithm might perform on new data. This was done for simplicity, but ideally, you could separate your data into train and test datasets for training and evaluation of your model.

You can evaluate your model on your training dataset using the `evaluate()` function and pass it the same input and output used to train the model. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy. The `evaluate()` function will return a list with two values. The first will be the loss of the model on the dataset, and the second will be the accuracy of the model on the dataset. You are only interested in reporting the accuracy so ignore the loss value.

```
...
# evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 7.6: Evaluate the neural network model on the dataset

7.6 Tie It All Together

You have just seen how you can easily create your first neural network model in Keras. Let's tie it all together into a complete code example.

```
# first neural network with keras tutorial
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
# define the keras model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
# evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f' % (accuracy*100))
```

Listing 7.7: Complete working example of your first neural network in Keras

You can copy all the code into your Python file and save it as "`keras_first_network.py`" in the same directory as your data file "`pima-indians-diabetes.csv`". You can then run the Python file as a script from your command line (command prompt) as follows:

```
python keras_first_network.py
```

Listing 7.8: Running a Python script

Running this example, you should see a message for each of the 150 epochs, printing the loss and accuracy, followed by the final evaluation of the trained model on the training dataset. It takes about 10 seconds to execute on my workstation running on the CPU. Ideally, you would like the loss to go to zero and the accuracy to go to 1.0 (e.g., 100%). This is not possible for any but the most trivial machine learning problems. Instead, you will always have some error in your model. The goal is to choose a model configuration and training configuration that achieve the lowest loss and highest accuracy possible for a given dataset.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
...
77/77 [==============================] - 0s 334us/step - loss: 0.4753 - accuracy: 0.7630
Epoch 147/150
77/77 [==============================] - 0s 336us/step - loss: 0.4794 - accuracy: 0.7565
Epoch 148/150
77/77 [==============================] - 0s 330us/step - loss: 0.4786 - accuracy: 0.7630
Epoch 149/150
77/77 [==============================] - 0s 327us/step - loss: 0.4777 - accuracy: 0.7669
Epoch 150/150
77/77 [==============================] - 0s 337us/step - loss: 0.4806 - accuracy: 0.7721
24/24 [==============================] - 0s 368us/step - loss: 0.4675 - accuracy: 0.7786
Accuracy: 77.86
```

Output 7.2: Output of running your first neural network in Keras

Neural networks are stochastic algorithms, meaning that the same algorithm on the same data can train a different model with different skill each time the code is run. This is a feature, not a bug. The variance in the performance of the model means that to get a reasonable approximation of how well your model is performing, you may need to fit it many times and calculate the average of the accuracy scores. For example, below are the accuracy scores from re-running the example five times:

```
Accuracy: 75.00
Accuracy: 77.73
Accuracy: 77.60
Accuracy: 78.12
Accuracy: 76.17
```

Output 7.3: Output of running the model five different times

You can see that all accuracy scores are around 77%, and the average is 76.924%.

7.7 Make Predictions

You can adapt the above example and use it to generate predictions on the training dataset, pretending it is a new dataset you have not seen before. Making predictions is as easy as calling the `predict()` function on the model. You are using a sigmoid activation function on the output layer, so the predictions will be a probability in the range between 0 and 1. You can easily convert them into a crisp binary prediction for this classification task by rounding them. For example:

```
...
# make probability predictions with the model
predictions = model.predict(X)
# round predictions
rounded = [round(x[0]) for x in predictions]
```

Listing 7.9: Example of predicting probabilities for each example

Alternately, you can convert the probability into 0 or 1 to predict crisp classes directly; for example:

```
...
# make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)
```

Listing 7.10: Example of prediction class labels for each example

The complete example below makes predictions for each example in the dataset, then prints the input data, predicted class, and expected class for the first five examples in the dataset.

```
# first neural network with keras make predictions
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
```

```
# define the keras model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10, verbose=0)
# make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)
# summarize the first 5 cases
for i in range(5):
    print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

Listing 7.11: Complete working example of fitting a model in Keras and using it to make predictions

Running the example does not show the progress bar as before, as the verbose argument has been set to 0. After the model is fit, predictions are made for all examples in the dataset, and the input rows and predicted class value for the first five examples is printed and compared to the expected class value. You can see that most rows are correctly predicted. In fact, you can expect about 76.9% of the rows to be correctly predicted based on your estimated performance of the model in the previous section.

```
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] => 0 (expected 1)
[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] => 0 (expected 0)
[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] => 1 (expected 1)
[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] => 0 (expected 0)
[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] => 1 (expected 1)
```

Output 7.4: Output of making predictions with Keras

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

7.8 Further Reading

Are you looking for some more Deep Learning tutorials with Python and Keras? Take a look at some of these:

Books

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
https://www.amazon.com/dp/0262035618
(Online version at http://www.deeplearningbook.org).

APIs

Keras official homepage.
https://keras.io
Keras API Reference.
https://keras.io/api/

7.9 Summary

In this chapter, you discovered how to create your first neural network model using the powerful Keras Python library for deep learning. Specifically, you learned the six key steps in using Keras to create a neural network or deep learning model step-by-step, including:

- ▷ How to load data
- ▷ How to define a neural network in Keras
- ▷ How to compile a Keras model using the efficient numerical backend
- ▷ How to train a model on data
- ▷ How to evaluate a model on data
- ▷ How to make predictions with the model

Now you have your first Keras model created and you will repeat this work flow for all deep learning projects. In the next chapter, we will look closer into the evaluation step.

Evaluate the Performance of Deep Learning Models

8

There are a lot of decisions to make when designing and configuring your deep learning models. Most of these decisions must be resolved empirically through trial and error and evaluating them on real data. As such, it is critically important to have a robust way to evaluate the performance of your neural networks and deep learning models. In this chapter, you will discover a few ways to evaluate model performance using Keras. After completing this lesson, you will know:

- ▷ How to evaluate a Keras model using an automatic verification dataset.
- ▷ How to evaluate a Keras model using a manual verification dataset.
- ▷ How to evaluate a Keras model using k-fold cross-validation.

Let's get started.

Overview

This chapter is in three parts; they are:

- ▷ Empirically Evaluate Network Configurations
- ▷ Data Splitting
- ▷ Manual k-Fold Cross-Validation

8.1 Empirically Evaluate Network Configurations

You must make a myriad of decisions when designing and configuring your deep learning models. Many of these decisions can be resolved by copying the structure of other people's networks and using heuristics. Ultimately, the best technique is to actually design small experiments and empirically evaluate problems using real data. This includes high-level decisions like the number, size, and type of layers in your network. It also includes lower-level decisions like the choice of the loss function, activation functions, optimization procedure, and the number of epochs.

Deep learning is often used on problems that have very large datasets. That is tens of thousands or hundreds of thousands of instances. As such, you need to have a robust test harness that allows you to estimate the performance of a given configuration on unseen data and reliably compare the performance to other configurations.

8.2 Data Splitting

The large amount of data and the complexity of the models require very long training times. As such, it is typical to separate data into training and test datasets or training and validation datasets. Keras provides two convenient ways of evaluating your deep learning algorithms this way:

1. Use an automatic verification dataset
2. Use a manual verification dataset

Use an Automatic Verification Dataset

Keras can separate a portion of your training data into a validation dataset and evaluate the performance of your model on that validation dataset in each epoch. You can do this by setting the `validation_split` argument on the `fit()` function to a percentage of the size of your training dataset. For example, a reasonable value might be 0.2 or 0.33 for 20% or 33% of your training data held back for validation. The example below demonstrates the use of an automatic validation dataset on a small binary classification problem using the Pima Indians onset of diabetes dataset (see Section 7.1).

```python
# MLP with automatic validation set
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy
# fix random seed for reproducibility
numpy.random.seed(7)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=150, batch_size=10)
```

Listing 8.1: Evaluate a neural network using an automatic validation set

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example, you can see that the verbose output on each epoch shows the loss and accuracy on both the training dataset and the validation dataset.

```
...
Epoch 145/150
514/514 [==============================] - 0s - loss: 0.5252 - acc: 0.7335 - val_loss: 0.5489 - val_
Epoch 146/150
514/514 [==============================] - 0s - loss: 0.5198 - acc: 0.7296 - val_loss: 0.5918 - val_
Epoch 147/150
514/514 [==============================] - 0s - loss: 0.5175 - acc: 0.7335 - val_loss: 0.5365 - val_
Epoch 148/150
514/514 [==============================] - 0s - loss: 0.5219 - acc: 0.7354 - val_loss: 0.5414 - val_
Epoch 149/150
514/514 [==============================] - 0s - loss: 0.5089 - acc: 0.7432 - val_loss: 0.5417 - val_
Epoch 150/150
514/514 [==============================] - 0s - loss: 0.5148 - acc: 0.7490 - val_loss: 0.5549 - val_
```

Output 8.1: Output of evaluating a neural network using an automatic validation set

Use a Manual Verification Dataset

Keras also allows you to manually specify the dataset to use for validation during training. In this example, you can use the handy `train_test_split()` function from the Python scikit-learn machine learning library to separate your data into a training and test dataset. Use 67% for training and the remaining 33% of the data for validation. The validation dataset can be specified to the `fit()` function in Keras by the `validation_data` argument. It takes a tuple of the input and output datasets.

```python
# MLP with manual validation set
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# split into 67% for train and 33% for test
X_train, X_test, y_train, y_test = train_test_split(X, Y,
                                                    test_size=0.33, random_state=seed)
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test,y_test), epochs=150, batch_size=10)
```

Listing 8.2: Evaluate a neural network using a manual validation set

Like before, running the example provides verbose output of training that includes the loss and accuracy of the model on both the training and validation datasets for each epoch.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
...
Epoch 145/150
514/514 [==============================] - 0s - loss: 0.4847 - acc: 0.7704 - val_loss: 0.5668 - val_
Epoch 146/150
514/514 [==============================] - 0s - loss: 0.4853 - acc: 0.7549 - val_loss: 0.5768 - val_
Epoch 147/150
514/514 [==============================] - 0s - loss: 0.4864 - acc: 0.7743 - val_loss: 0.5604 - val_
Epoch 148/150
514/514 [==============================] - 0s - loss: 0.4831 - acc: 0.7665 - val_loss: 0.5589 - val_
Epoch 149/150
514/514 [==============================] - 0s - loss: 0.4961 - acc: 0.7782 - val_loss: 0.5663 - val_
Epoch 150/150
514/514 [==============================] - 0s - loss: 0.4967 - acc: 0.7588 - val_loss: 0.5810 - val_
```

Output 8.2: Output of evaluating a neural network using an manual validation set

8.3 Manual *k*-Fold Cross-Validation

The gold standard for machine learning model evaluation is k-fold cross-validation. It provides a robust estimate of the performance of a model on unseen data. It does this by splitting the training dataset into k subsets, taking turns training models on all subsets except one, which is held out, and evaluating model performance on the held-out validation dataset. The process is repeated until all subsets are given an opportunity to be the held-out validation set. The performance measure is then averaged across all models that are created.

It is important to understand that cross-validation means estimating a model design (e.g., 3-layer vs. 4-layer neural network) rather than a specific fitted model. You do not want to use a specific dataset to fit the models and compare the result since this may be due to that particular dataset fitting better on one model design (as known as *overfitting*). Instead, you want to use multiple datasets to fit, resulting in multiple fitted models of the same design, taking the average performance measure for comparison.

Cross-validation is often not used for evaluating deep learning models because of the greater computational expense. For example, k-fold cross-validation is often used with 5 or 10 folds. As such, 5 or 10 models must be constructed and evaluated, significantly adding to the evaluation time of a model. Nevertheless, when the problem is small enough or if you have sufficient computing resources, k-fold cross-validation can give you a less-biased estimate of the performance of your model.

In the example below, you will use the handy `StratifiedKFold` class from the scikit-learn Python machine learning library to split the training dataset into 10 folds. The folds are stratified, meaning that the algorithm attempts to balance the number of instances of each class in each fold. The example creates and evaluates 10 models using the 10 splits of the data and collects all the scores. The verbose output for each epoch is turned off by passing `verbose=0` to the `fit()` and `evaluate()` functions on the model. The performance is printed for each model, and it is stored. The average and standard deviation of the model performance are then printed at the end of the run to provide a robust estimate of model accuracy.

```python
# MLP for Pima Indians Dataset with 10-fold cross validation
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import StratifiedKFold
import numpy as np
# fix random seed for reproducibility
seed = 7
np.random.seed(seed)
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# define 10-fold cross validation test harness
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
cvscores = []
for train, test in kfold.split(X, Y):
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # Fit the model
    model.fit(X[train], Y[train], epochs=150, batch_size=10, verbose=0)
    # evaluate the model
    scores = model.evaluate(X[test], Y[test], verbose=0)
    print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
    cvscores.append(scores[1] * 100)

print("%.2f%% (+/- %.2f%%)" % (np.mean(cvscores), np.std(cvscores)))
```

Listing 8.3: Evaluate a neural network using scikit-learn

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example will take less than a minute and will produce the following output:

```
acc: 77.92%
acc: 68.83%
acc: 72.73%
acc: 64.94%
acc: 77.92%
acc: 35.06%
acc: 74.03%
acc: 68.83%
acc: 34.21%
acc: 72.37%
64.68% (+/- 15.50%)
```

Output 8.3: Output of evaluating a neural network using scikit-learn

Notice that we had to re-create the model each loop then fit and evaluate it with the data for the fold. In chapter 13, we will look at how we can use Keras models natively with the scikit-learn machine learning library.

8.4 Summary

In this chapter, you discovered the importance of having a robust way to estimate the performance of your deep learning models on unseen data. You learned three ways that you can estimate the performance of your deep learning models in Python using the Keras library:

- ▷ Automatically splitting a training dataset into train and validation datasets
- ▷ Manually and explicitly defining a training and validation dataset
- ▷ Evaluating performance using k-fold cross-validation, the gold standard technique

In the next chapter, we will see another example of using Keras to build a neural network.

9 Project: Multiclass Classification of Iris Species

In this chapter, you will discover how to use Keras to develop and evaluate neural network models for multiclass classification problems. After completing this chapter, you will know:

- ▷ How to load data from CSV and make it available to Keras
- ▷ How to prepare multiclass classification data for modeling with neural networks
- ▷ How to evaluate Keras neural network models with scikit-learn

Let's get started.

Overview

This chapter is in six parts; they are:

- ▷ Problem Description
- ▷ Import Classes and Functions
- ▷ Load the Dataset
- ▷ Encode the Output Variable
- ▷ Evaluate the Model with k-Fold Cross-Validation
- ▷ Complete Example

9.1 Problem Description

In this chapter, you will use the standard machine learning problem called the iris flowers dataset. This dataset is well studied and makes a good problem for practicing on neural networks because all four input variables are numeric and have the same scale in centimeters. Each instance describes the properties of an observed flower's measurements, and the output variable is a specific iris species. The attributes for this dataset can be summarized as follows:

1. Sepal length in centimeters
2. Sepal width in centimeters
3. Petal length in centimeters

4. Petal width in centimeters
5. Class (the flower species)

This is a multiclass classification problem, meaning that there are more than two classes to be predicted. In fact, there are three flower species. This is an important problem for practicing with neural networks because the three class values require specialized handling. Below is a sample of the first five of the 150 instances:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
```

Output 9.1: Sample of the iris flowers dataset

The iris flower dataset is a well-studied problem, and as such, you can expect to achieve a model accuracy in the range of 95% to 97%. This provides a good target to aim for when developing your models. The dataset is provided in the bundle of sample code provided with this book. You can also download the iris flowers dataset from the UCI Machine Learning repository and place it in your current working directory with the filename `iris.csv`. You can learn more about the iris flower classification dataset on the UCI Machine Learning Repository page.

9.2 Import Classes and Functions

You can begin by importing all the classes and functions you will need in this chapter. This includes both the functionality you require from Keras and the data loading from pandas, as well as data preparation and model evaluation from scikit-learn and SciKeras.

```python
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
...
```

Listing 9.1: Import classes and functions

9.3 Load the Dataset

The dataset can be loaded directly. Because the output variable contains strings, it is easiest to load the data using pandas. You can then split the attributes (columns) into input variables (X) and output variables (Y).

```
...
# load dataset
dataframe = pd.read_csv("iris.csv", header=None)
dataset = dataframe.values
X = dataset[:,0:4].astype(float)
Y = dataset[:,4]
```

Listing 9.2: Load dataset and separate into input and output variables

9.4 Encode the Output Variable

The output variable contains three different string values. When modeling multiclass classification problems using neural networks, it is good practice to reshape the output attribute from a vector of class labels to a matrix of a Boolean for each class and whether a given instance is in that class. This is called one-hot encoding or creating dummy variables from a categorical variable. For example, in this problem, the three class values are `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`. If you had the three observations:

```
Iris-setosa
Iris-versicolor
Iris-virginica
```

Output 9.2: Three classes in the iris dataset

You can turn this into a one-hot encoded binary matrix for each data instance that would look like this:

```
Iris-setosa,    Iris-versicolor,    Iris-virginica
1,              0,                  0
0,              1,                  0
0,              0,                  1
```

Output 9.3: One-hot encoding of the classes in the iris dataset

You can first encode the strings consistently to integers using the scikit-learn class `LabelEncoder`. Then convert the vector of integers to a one-hot encoding using the Keras function `to_categorical()`.

```
...
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one-hot encoded)
dummy_y = to_categorical(encoded_Y)
```

Listing 9.3: One-hot encoding of iris dataset output variable

9.5 Define the Neural Network Model

The Keras library provides wrapper classes to allow you to use neural network models developed with Keras in scikit-learn. There is a `KerasClassifier` class in SciKeras that can be used as an estimator in scikit-learn, the base type of model in the library. The `KerasClassifier` takes the name of a function as an argument. This function must return the constructed neural network model, ready for training.

Below is a function that will create a baseline neural network for the iris classification problem. It creates a simple, fully connected network with one hidden layer that contains eight neurons. The hidden layer uses a rectifier activation function which is a good practice. Because you used a one-hot encoding for your iris dataset, the output layer must create three output values, one for each class. The output value with the largest value will be taken as the class predicted by the model. The network topology of this simple one-layer neural network can be summarized as follows:

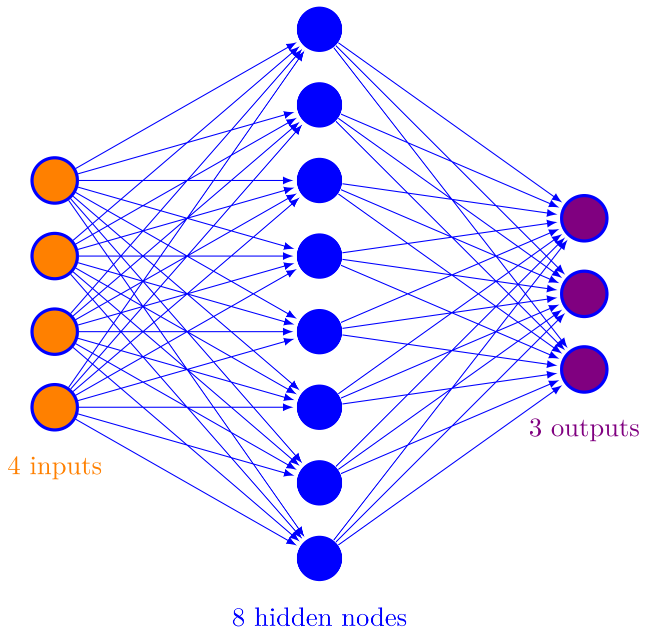


Figure 9.1: Example network structure

Note that a "`softmax`" activation function was used in the output layer. This ensures the output values are in the range of 0 and 1 and may be used as predicted probabilities. Finally, the network uses the efficient Adam gradient descent optimization algorithm with a logarithmic loss function, which is called "`categorical_crossentropy`" in Keras.

```python
...
# define baseline model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(8, input_shape=(4,), activation='relu'))
    model.add(Dense(3, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Listing 9.4: Define and compile the neural network model

You can now create your `KerasClassifier` for use in scikit-learn. You can also pass arguments in the construction of the `KerasClassifier` class that will be passed on to the `fit()` function internally used to train the neural network. Here, you pass the number of epochs as 200 and batch size as 5 to use when training the model. Debugging is also turned off when training by setting `verbose` to 0.

```
...
estimator = KerasClassifier(model=baseline_model, epochs=200, batch_size=5, verbose=0)
```

Listing 9.5: Create wrapper for neural netowrk model for use in scikit-learn

9.6 Evaluate the Model with *k*-Fold Cross-Validation

You can now evaluate the neural network model on our training data. The scikit-learn library has excellent capability to evaluate models using a suite of techniques. The gold standard for evaluating machine learning models is *k*-fold cross-validation. First, define the model evaluation procedure. Here, you set the number of folds to 10 (an excellent default) and shuffle the data before partitioning it.

```
...
kfold = KFold(n_splits=10, shuffle=True)
```

Listing 9.6: Prepare cross-validation

Now, you can evaluate your model (`estimator`) on your dataset (`X` and `dummy_y`) using a 10-fold cross-validation procedure (`KFold`). Evaluating the model only takes approximately 10 seconds and returns an object that describes the evaluation of the ten constructed models for each of the splits of the dataset.

```
...
results = cross_val_score(estimator, X, dummy_y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 9.7: Evaluate the neural network model

9.7 Complete Example

You can tie all of this together into a single program that you can save and run as a script:

```
# multi-class classification with Keras
import pandas
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
# load dataset
dataframe = pandas.read_csv("iris.csv", header=None)
dataset = dataframe.values
X = dataset[:,0:4].astype(float)
Y = dataset[:,4]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one-hot encoded)
dummy_y = to_categorical(encoded_Y)

# define baseline model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(8, input_shape=(4,), activation='relu'))
    model.add(Dense(3, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

estimator = KerasClassifier(model=baseline_model, epochs=200, batch_size=5, verbose=0)
kfold = KFold(n_splits=10, shuffle=True)
results = cross_val_score(estimator, X, dummy_y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 9.8: Multilayer perceptron model for iris flower problem

The results are summarized as both the mean and standard deviation of the model accuracy on the dataset. This is a reasonable estimation of the performance of the model on unseen data. It is also within the realm of known top results for this problem.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Accuracy: 97.33% (4.42%)
```

Output 9.4: Estimated accuracy of neural network model on the Iris dataset

9.8 Further Readings

This section provides more resources on the topic if you are looking to go deeper.

Online Resources

Iris Data Set. UCI Machine Learning Repository.
https://archive.ics.uci.edu/ml/datasets/Iris
SciKeras documentation.
https://www.adriangb.com/scikeras/stable/
KFold cross-validator. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

9.9 Summary

In this chapter, you discovered how to develop and evaluate a neural network using the Keras Python library for deep learning. By completing this chapter, you learned:

- ▷ How to load data and make it available to Keras
- ▷ How to prepare multiclass classification data for modeling using one-hot encoding
- ▷ How to use Keras neural network models with scikit-learn
- ▷ How to define a neural network using Keras for multiclass classification
- ▷ How to evaluate a Keras neural network model using scikit-learn with k-fold cross-validation

In the next chapter, you will see how Keras can be used for classification of only two classes.

10 Project: Binary Classification of Sonar Returns

In this chapter you will discover how to effectively use the Keras library in your machine learning project by working through a binary classification project step-by-step. After completing this chapter, you will know:

- ▷ How to load training data and make it available to Keras
- ▷ How to design and train a neural network for tabular data
- ▷ How to evaluate the performance of a neural network model in Keras on unseen data
- ▷ How to perform data preparation to improve skill when using neural networks
- ▷ How to tune the topology and configuration of neural networks in Keras

Let's get started.

Overview

This chapter is in four parts; they are:

- ▷ Sonar Object Classification Dataset
- ▷ Baseline Neural Network Model Performance
- ▷ Improve the Baseline Model with Data Preparation
- ▷ Tuning Layers and Number of Neurons in the Model

10.1 Sonar Object Classification Dataset

The dataset you will use in this chapter is the Sonar dataset. This is a dataset that describes sonar chirp returns bouncing off different surfaces. The 60 input variables are the strength of the returns at different angles. It is a binary classification problem that requires a model to differentiate rocks from metal cylinders.

It is a well-understood dataset. All the variables are continuous and generally in the range of 0 to 1. The output variable is a string "`M`" for mine and "`R`" for rock, which will need to be converted to integers 1 and 0 as neural networks can only output numbers. The dataset is in the bundle of source code provided with this book. Alternatively, you can download[1] the dataset and place it in your working directory with the filename `sonar.csv`.

A benefit of using this dataset is that it is a standard benchmark problem. This means that we have some idea of the expected skill of a good model. Using cross-validation, a neural network should be able to achieve a performance of around 84% with an upper bound on accuracy for custom models at around 88%. You can learn more about this dataset on the UCI Machine Learning repository.

10.2 Baseline Neural Network Model Performance

Let's create a baseline model and result for this problem. You will start by importing all the classes and functions you will need.

```
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
...
```

Listing 10.1: Import classes and functions

Now, you can load the dataset using Pandas and split the columns into 60 input variables (`X`) and one output variable (`Y`). Use Pandas to load the data because it easily handles strings (the output variable), whereas attempting to load the data directly using NumPy would be more difficult.

```
...
# load dataset
dataframe = pd.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
```

Listing 10.2: Load the dataset and separate into input and output variables

The output variable is string values. You must convert them into integer values 0 and 1. You can do this using the `LabelEncoder` class from scikit-learn. This class will model the encoding required using the entire dataset via the `fit()` function, then apply the encoding to create a new output variable using the `transform()` function.

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/sonar.csv

```
...
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
```

Listing 10.3: Label encode output variables

You are now ready to create your neural network model using Keras. You will use scikit-learn to evaluate the model using stratified k-fold cross-validation. This is a resampling technique that will provide an estimate of the performance of the model. It does this by splitting the data into k parts and training the model on all parts except one, which is held out as a test set to evaluate the performance of the model. This process is repeated k times, and the average score across all constructed models is used as a robust estimate of performance. It is stratified, meaning that it will look at the output values and attempt to balance the number of instances that belong to each class in the k splits of the data.

To use Keras models with scikit-learn, you must use the `KerasClassifier` wrapper from the SciKeras module. This class takes a function that creates and returns our neural network model. It also takes arguments that it will pass along to the call to `fit()`, such as the number of epochs and the batch size.

Let's start by defining the function that creates your baseline model. Your model will have a single, fully connected hidden layer with the same number of neurons as input variables. This is a good default starting point when creating neural networks.

The weights are initialized using a small Gaussian random number. The Rectifier activation function is used. The output layer contains a single neuron in order to make predictions. It uses the sigmoid activation function in order to produce a probability output in the range of 0 to 1 that can easily and automatically be converted to crisp class values.

Finally, you will use the logarithmic loss function (`binary_crossentropy`) during training, the preferred loss function for binary classification problems. The model also uses the efficient Adam optimization algorithm for gradient descent, and accuracy metrics will be collected when the model is trained.

```
# baseline model
def create_baseline():
    # create model
    model = Sequential()
    model.add(Dense(60, input_shape=(60,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Listing 10.4: Define and compile baseline model

Now, it is time to evaluate this model using stratified cross-validation in the scikit-learn framework. Pass the number of training epochs to the `KerasClassifier`, again using reasonable default values. Verbose output is also turned off, given that the model will be created ten times for the 10-fold cross-validation being performed.

```
...
# evaluate model with standardized dataset
estimator = KerasClassifier(model=create_baseline, epochs=100, batch_size=5, verbose=0)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.5: Fit and evaluate baseline model

After tying this together, the complete example is listed below.

```
# Binary Classification with Sonar Dataset: Baseline
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
# load dataset
dataframe = read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# baseline model
def create_baseline():
    # create model
    model = Sequential()
    model.add(Dense(60, input_shape=(60,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# evaluate model with standardized dataset
estimator = KerasClassifier(model=create_baseline, epochs=100, batch_size=5, verbose=0)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.6: Multilayer perceptron model for sonar problem

Running this code produces the following output showing the mean and standard deviation of the estimated accuracy of the model on unseen data.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Baseline: 81.68% (7.26%)
```

Output 10.1: Sample output from fitting and evaluating the baseline model

This is an excellent score without doing any hard work.

10.3 Improve the Baseline Model with Data Preparation

It is a good practice to prepare your data before modeling. Neural network models are especially suitable for having consistent input values, both in scale and distribution. Standardization is an effective data preparation scheme for tabular data when building neural network models. This is where the data is rescaled such that the mean value for each attribute is 0, and the standard deviation is 1. This preserves Gaussian and Gaussian-like distributions while normalizing the central tendencies for each attribute. You can use scikit-learn to perform the standardization of your sonar dataset using the `StandardScaler` class.

Rather than performing the standardization on the entire dataset, it is good practice to train the standardization procedure on the training data within the pass of a cross-validation run, and use the trained standardization instance to prepare the unseen test fold. This makes standardization a step in model preparation in the cross-validation process. It prevents the algorithm from having knowledge of unseen data during evaluation, knowledge that might be passed from the data preparation scheme like a crisper distribution.

You can achieve this in scikit-learn using a `Pipeline` class. The pipeline is a wrapper that executes one or more models within a pass of the cross-validation procedure. Here, you can define a pipeline with the `StandardScaler` followed by your neural network model.

```python
...
# evaluate baseline model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_baseline,
                                          epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.7: Preprocess the sonar data with standardization

After tying this together, the complete example is listed below.

```python
# Binary Classification with Sonar Dataset: Standardized
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# baseline model
def create_baseline():
    # create model
    model = Sequential()
    model.add(Dense(60, input_shape=(60,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# evaluate baseline model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_baseline,
                                          epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.8: Update experiment to use data standardization

Running this example provides the results below.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

You now see a small but very nice lift in the mean accuracy.

```
Standardized: 84.56% (5.74%)
```

Output 10.2: Sample output from update using data standardization

10.4 Tuning Layers and Number of Neurons in the Model

There are many things to tune on a neural network, such as weight initialization, activation functions, optimization procedure, and so on. One aspect that may have an outsized effect is the structure of the network itself, called the network topology. In this section, you will look at two experiments on the structure of the network: making it smaller and making it larger. These are good experiments to perform when tuning a neural network on your problem.

Evaluate a Smaller Network

Note that there is likely a lot of redundancy in the input variables for this problem. The data describes the same signal from different angles. Perhaps some of those angles are more relevant than others. So you can force a type of feature extraction by the network by restricting the representational space in the first hidden layer.

In this experiment, you will take your baseline model with 60 neurons in the hidden layer and reduce it by half to 30. This will pressure the network during training to pick out the most important structure in the input data to model.

You will also standardize the data as in the previous experiment with data preparation and try to take advantage of the slight lift in performance.

```python
...
# smaller model
def create_smaller():
    # create model
    model = Sequential()
    model.add(Dense(30, input_shape=(60,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_smaller,
                                          epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Smaller: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.9: Creating a smaller network topology

After tying this together, the complete example is listed below.

```python
# Binary Classification with Sonar Dataset: Standardized Smaller
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("sonar.csv", header=None)
dataset = dataframe.values
```

```
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# smaller model
def create_smaller():
    # create model
    model = Sequential()
    model.add(Dense(30, input_shape=(60,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_smaller,
                                          epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Smaller: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.10: Update to use a smaller network topology

Running this example provides the following result. You can see that you have a very slight boost in the mean estimated accuracy and an important reduction in the standard deviation (average spread) of the accuracy scores for the model. This is a great result because we are doing slightly better with a network half the size, which, in turn, takes half the time to train.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Smaller: 86.04% (4.00%)
```

Output 10.3: Sample output from using a smaller network topology

Evaluate a Larger Network

A neural network topology with more layers offers more opportunities for the network to extract key features and recombine them in useful nonlinear ways. You can easily evaluate whether adding more layers to the network improves the performance by making another small tweak to the function used to create our model. Here, you add one new layer (one line) to the network that introduces another hidden layer with 30 neurons after the first hidden layer. Your network now has the topology:

```
60 inputs -> [60 -> 30] -> 1 output
```

Output 10.4: Summary of new network topology

The idea here is that the network is given the opportunity to model all input variables before being bottlenecked and forced to halve the representational capacity, much like you did in the experiment above with the smaller network. Instead of squeezing the representation of the inputs themselves, you have an additional hidden layer to aid in the process.

```
...
# larger model
def create_larger():
    # create model
    model = Sequential()
    model.add(Dense(60, input_shape=(60,), activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_larger,
                                          epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Larger: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.11: Creating a larger network toplogy

After tying this together, the complete example is listed below.

```
# Binary Classification with Sonar Dataset: Standardized Larger
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
```

```
# larger model
def create_larger():
    # create model
    model = Sequential()
    model.add(Dense(60, input_shape=(60,), activation='relu'))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_larger,
                                          epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Larger: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.12: Update to use a larger network topology

Running this example produces the results below. You can see that you do not get a lift in the model performance. This may be statistical noise or a sign that further training is needed.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Larger: 83.14% (4.52%)
```

Output 10.5: Sample output from using a larger network topology

With further tuning of aspects like the optimization algorithm and the number of training epochs, it is expected that further improvements are possible. What is the best score that you can achieve on this dataset?

10.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Online Resources

Connectionist Bench (Snoar, Mines vs. Rocks) Data Set. UCI Machine Learning Repository.
https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)

SciKeras documentation.
https://www.adriangb.com/scikeras/stable/

KFold cross-validator. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

10.6 Summary

In this chapter, you discovered the Keras deep learning library in Python. You learned how you can work through a binary classification problem step-by-step with Keras, specifically:

- How to load and prepare data for use in Keras
- How to create a baseline neural network model
- How to evaluate a Keras model using scikit-learn and stratified k-fold cross-validation
- How data preparation schemes can lift the performance of your models
- How experiments adjusting the network topology can lift model performance

In the next chapter, you will see Keras can build models to solve not only classification, but also regression problems.

Project: Regression of Boston House Prices

11

In this chapter, you will discover how to develop and evaluate neural network models using Keras for a regression problem. After completing this chapter, you will know:

- ▷ How to load a CSV dataset and make it available to Keras
- ▷ How to create a neural network model with Keras for a regression problem
- ▷ How to use scikit-learn with Keras to evaluate models using cross-validation
- ▷ How to perform data preparation in order to improve skill with Keras models
- ▷ How to tune the network topology of models with Keras

Let's get started.

Overview

This chapter is in four parts; they are:

- ▷ Problem Description
- ▷ Develop a Baseline Neural Network Model
- ▷ Modeling the Standardized Dataset
- ▷ Tune the Neural Network Topology

11.1 Problem Description

The problem that we will look at in this chapter is the Boston house price dataset. The dataset describes 13 numerical properties of houses in Boston suburbs and is concerned with modeling the price of houses in those suburbs (the 14th property) in thousands of dollars. As such, this is a regression predictive modeling problem. The full list of attributes in this dataset are as follows:

1. `CRIM`: per capita crime rate by town
2. `ZN`: proportion of residential land zoned for lots over 25,000 sq.ft.
3. `INDUS`: proportion of non-retail business acres per town

4. `CHAS`: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. `NOX`: nitric oxides concentration (parts per 10 million)
6. `RM`: average number of rooms per dwelling
7. `AGE`: proportion of owner-occupied units built prior to 1940
8. `DIS`: weighted distances to five Boston employment centers
9. `RAD`: index of accessibility to radial highways
10. `TAX`: full-value property-tax rate per $10,000
11. `PTRATIO`: pupil-teacher ratio by town
12. `B`: $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
13. `LSTAT`: % lower status of the population
14. `MEDV`: Median value of owner-occupied homes in $1000s

This is a well-studied problem in machine learning. It is convenient to work with because all the input and output attributes are numerical, and there are 506 instances to work with. A sample of the first 5 of the 506 rows in the dataset is provided below:

```
0.00632  18.00   2.310  0  0.5380  6.5750  65.20  4.0900   1  296.0  15.30 396.90   4.98  24.00
0.02731   0.00   7.070  0  0.4690  6.4210  78.90  4.9671   2  242.0  17.80 396.90   9.14  21.60
0.02729   0.00   7.070  0  0.4690  7.1850  61.10  4.9671   2  242.0  17.80 392.83   4.03  34.70
0.03237   0.00   2.180  0  0.4580  6.9980  45.80  6.0622   3  222.0  18.70 394.63   2.94  33.40
0.06905   0.00   2.180  0  0.4580  7.1470  54.20  6.0622   3  222.0  18.70 396.90   5.33  36.20
```

Output 11.1: Sample of the Boston house price dataset

The dataset is available in the bundle of source code provided with this book. Alternatively, you can download[1] this dataset and save it to your current working directory with the file name `housing.csv`. Reasonable performance for models evaluated using Mean Squared Error (MSE) is around 20 in thousands of dollars squared (or $4,500 if you take the square root). This is a nice target to aim for with our neural network model.

11.2 Develop a Baseline Neural Network Model

In this section, you will create a baseline neural network model for the regression problem. Let's start by importing all the functions and objects you will need for this chapter.

```python
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
...
```

Listing 11.1: Import classes and functions

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.data

You can now load your dataset from a file in the local directory. The dataset is, in fact, not in CSV format in the UCI Machine Learning Repository. The attributes are instead separated by whitespace. You can load this easily using the Pandas library. Then split the input (X) and output (Y) attributes, making them easier to model with Keras and scikit-learn.

```
...
# load dataset
dataframe = pd.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
```

Listing 11.2: Load dataset and separate into input and output variables

You can create Keras models and evaluate them with scikit-learn using handy wrapper objects provided by the SciKeras library. This is desirable, because scikit-learn excels at evaluating models and will allow you to use powerful data preparation and model evaluation schemes with very few lines of code. The Keras wrappers require a function as an argument. This function you must define is responsible for creating the neural network model to be evaluated.us

Below, you will define the function to create the baseline model to be evaluated. It is a simple model with a single fully connected hidden layer with the same number of neurons as input attributes (13). The network uses good practices such as the rectifier activation function for the hidden layer. No activation function is used for the output layer because it is a regression problem, and you are interested in predicting numerical values directly without transformation.

The efficient ADAM optimization algorithm is used, and a mean squared error loss function is optimized. This will be the same metric you will use to evaluate the performance of the model. It is a desirable metric because taking the square root of an error value allows you to directly understand in the context of the problem with the units in thousands of dollars.

```
...
# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_shape=(13,),
                    kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

Listing 11.3: Define and compile a baseline neural network model

The SciKeras wrapper object used in scikit-learn as a regression estimator is called `KerasRegressor`. You create an instance and pass it both the name of the function to create the neural network model and some parameters to pass along to the `fit()` function of the model later, such as the number of epochs and batch size. Both of these are set to sensible defaults.

```
...
# evaluate model
estimator = KerasRegressor(model=baseline_model, epochs=100, batch_size=5, verbose=0)
```

Listing 11.4: Prepare model wrapper for scikit-learn

The final step is to evaluate this baseline model. You will use 10-fold cross-validation to evaluate the model. MSE is the metric here but since `cross_val_score()` is to find the one with highest score, `neg_mean_squared_error` (negative of MSE) is used.

```
...
kfold = KFold(n_splits=10)
results = cross_val_score(estimator, X, Y, cv=kfold, scoring='neg_mean_squared_error')
print("Results: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 11.5: Evaluate baseline model

After tying this all together, the complete example is listed below.

```
# Regression Example With Boston Dataset: Baseline
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
# load dataset
dataframe = read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_shape=(13,),
                    kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
# evaluate model
estimator = KerasRegressor(model=baseline_model, epochs=100, batch_size=5, verbose=0)
kfold = KFold(n_splits=10)
results = cross_val_score(estimator, X, Y, cv=kfold, scoring='neg_mean_squared_error')
print("Baseline: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 11.6: Multilayer perceptron model for Boston house problem

Running this code gives you an estimate of the model's performance on the problem for unseen data. The result reports the mean squared error, including the average and standard deviation (average variance) across all 10 folds of the cross-validation evaluation.

Note: The mean squared error is negative because scikit-learn inverts so that the metric is maximized instead of minimized. You can ignore the sign of the result.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Baseline: -32.65 (23.33) MSE
```

Output 11.2: Sample output from evaluating the baseline model

11.3 Modeling the Standardized Dataset

An important concern with the Boston house price dataset is that the input attributes all vary in their scales because they measure different quantities. It is almost always good practice to prepare your data before modeling it using a neural network model. Continuing from the above baseline model, you can re-evaluate the same model using a standardized version of the input dataset.

You can use scikit-learn's `Pipeline` framework to perform the standardization during the model evaluation process within each fold of the cross-validation. This ensures that there is no data leakage from each test set cross-validation fold into the training data. The code below creates a scikit-learn `Pipeline` that first standardizes the dataset and then creates and evaluates the baseline neural network model.

```
...
# evaluate model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(model=baseline_model,
                                          epochs=50, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10)
results = cross_val_score(pipeline, X, Y, cv=kfold, scoring='neg_mean_squared_error')
print("Standardized: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 11.7: Evaluate model with standardized datset

After tying this together, the complete example is listed below.

```
# Regression Example With Boston Dataset: Standardized
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_shape=(13,),
                    kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
# evaluate model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(model=baseline_model,
                                         epochs=50, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10)
results = cross_val_score(pipeline, X, Y, cv=kfold, scoring='neg_mean_squared_error')
print("Standardized: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 11.8: Update to use a standardized dataset

Running the example provides an improved performance over the baseline model without standardized data, dropping the error.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Standardized: -29.54 (27.87) MSE
```

Output 11.3: Sample output from evaluating the model on the standardized datset

A further extension of this section would be to similarly apply a rescaling to the output variable, such as normalizing it to the range of 0 to 1 and using a sigmoid or similar activation function on the output layer to narrow output predictions to the same range.

11.4 Tune the Neural Network Topology

Many concerns can be optimized for a neural network model. Perhaps the point of biggest leverage is the structure of the network itself, including the number of layers and the number of neurons in each layer. In this section, you will evaluate two additional network topologies in an effort to further improve the performance of the model. You will look at both a deeper and a wider network topology.

Evaluate a Deeper Network Topology

One way to improve the performance of a neural network is to add more layers. This might allow the model to extract and recombine higher-order features embedded in the data. In this section, you will evaluate the effect of adding one more hidden layer to the model. This is as easy as defining a new function to create this deeper model, copied from your baseline model above. You can then insert a new line after the first hidden layer — in this case, with about half the number of neurons.

```
...
# define the model
def larger_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_shape=(13,),
              kernel_initializer='normal', activation='relu'))
    model.add(Dense(6, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

Listing 11.9: Making a deeper network topology

Your network topology now looks like this:

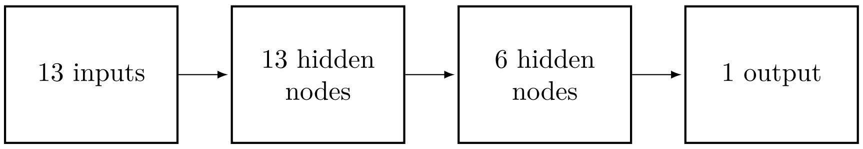


Figure 11.1: Summary of deeper network topology

You can evaluate this network topology in the same way as above, while also using the standardization of the dataset shown above to improve performance.

```
...
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(model=larger_model,
                                         epochs=50, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10)
results = cross_val_score(pipeline, X, Y, cv=kfold, scoring='neg_mean_squared_error')
print("Larger: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 11.10: Evaluate the house problem with standardization and deeper network

After tying this together, the complete example is listed below.

```
# Regression Example With Boston Dataset: Standardized and Larger
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define the model
def larger_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_shape=(13,),
                    kernel_initializer='normal', activation='relu'))
    model.add(Dense(6, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
# evaluate model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(model=larger_model,
                                         epochs=50, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10)
results = cross_val_score(pipeline, X, Y, cv=kfold, scoring='neg_mean_squared_error')
print("Larger: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 11.11: Evaluate the larger neural network model

Running this model shows a further improvement in MSE performance.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Larger: -22.83 (25.33) MSE
```

Output 11.4: Sample output from evaluating the deeper model

Evaluate a Wider Network Topology

Another approach to increasing the representational capacity of the model is to create a wider network. In this section, you will evaluate the effect of keeping a shallow network architecture and nearly doubling the number of neurons in the one hidden layer. Again, all you need to do is define a new function that creates your neural network model. Here, you will increase the number of neurons in the hidden layer compared to the baseline model from 13 to 20.

```
...
# define wider model
def wider_model():
    # create model
    model = Sequential()
    model.add(Dense(20, input_shape=(13,),
                    kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

Listing 11.12: Creating a wider model

Your network topology now looks like this:

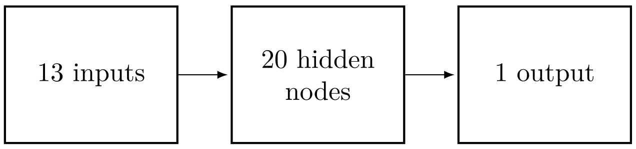


Figure 11.2: Summary of deeper network topology

You can evaluate the wider network topology using the same scheme as above:

```
...
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(model=wider_model,
                                         epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10)
results = cross_val_score(pipeline, X, Y, cv=kfold, scoring='neg_mean_squared_error')
print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 11.13: Evaluate a wider network with standardized data

After tying this together, the complete example is listed below.

```
# Regression Example With Boston Dataset: Standardized and Wider
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
```

```
dataframe = read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define wider model
def wider_model():
    # create model
    model = Sequential()
    model.add(Dense(20, input_shape=(13,),
                    kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
# evaluate model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(model=wider_model,
                                         epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n_splits=10)
results = cross_val_score(pipeline, X, Y, cv=kfold, scoring='neg_mean_squared_error')
print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 11.14: Evaluate the wider neural network model

Building the model reveals a further drop in error to about 21 thousand squared dollars. This is not a bad result for this problem.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Wider: -21.71 (24.39) MSE
```

Output 11.5: Sample output from evaluating the wider model

It might have been hard to guess that a wider network would outperform a deeper network on this problem. The results demonstrate the importance of empirical testing in developing neural network models.

11.5 Further Readings

This section provides more resources on the topic if you are looking to go deeper.

Online Resources

Boston Housing Dataset.
http://lib.stat.cmu.edu/datasets/boston

SciKeras documentation.
https://www.adriangb.com/scikeras/stable/
KFold cross-validator. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

11.6 Summary

In this chapter you discovered the Keras deep learning library for modeling regression problems. Through this chapter you learned how to develop and evaluate neural network models, including:

- How to load data and develop a baseline model.
- How to lift performance using data preparation techniques like standardization.
- How to design and evaluate networks with different varying topologies on a problem.

After these examples, you should understand what Keras can do. In the next chapter, we will see several syntax variations in Keras.

12 Three Ways to Build Models in Keras

If you've looked at how other people wrote Keras models on Github, you've probably noticed that there are some different ways to create models in Keras. There's the `Sequential` model, which allows you to define an entire model in a single line, usually with some line breaks for readability. Then, there's the functional interface that allows for more complicated model architectures, and there's also the `Model` subclass which helps reusability. This chapter will explore the different ways to create models in Keras, along with their advantages and drawbacks. This will equip you with the knowledge you need to create your own machine learning models in Keras. After you complete this chapter, you will learn:

- ▷ Different ways that Keras offers to build models
- ▷ How to use the `Sequential` class, functional interface, and subclassing `keras.Model` to build Keras models
- ▷ When to use the different methods to create Keras models

Let's get started!

Overview

This chapter is in three parts, covering the different ways to build machine learning models in Keras:

- ▷ Using the `Sequential` class
- ▷ Using Keras' functional interface
- ▷ Subclassing `keras.Model`

12.1 Using the Sequential class

The `Sequential` model is just as the name implies. It consists of a sequence of layers, one after the other. From the Keras documentation,

> A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor.
>
> — Keras documentation

It is a simple, easy-to-use way to start building your Keras model. To start, import TensorFlow and then the `Sequential` model:

```
import tensorflow as tf
from tensorflow.keras import Sequential
```

Listing 12.1: Import statements for using `Sequential` model

Then, you can start building your machine learning model by stacking various layers together. For this example, let's build a LeNet5 model with the classic CIFAR-10 image dataset as the input:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Flatten, Conv2D, MaxPool2D

model = Sequential([
          Input(shape=(32,32,3,)),
          Conv2D(6, (5,5), padding="same", activation="relu"),
          MaxPool2D(pool_size=(2,2)),
          Conv2D(16, (5,5), padding="same", activation="relu"),
          MaxPool2D(pool_size=(2, 2)),
          Conv2D(120, (5,5), padding="same", activation="relu"),
          Flatten(),
          Dense(units=84, activation="relu"),
          Dense(units=10, activation="softmax"),
      ])

model.summary()
```

Listing 12.2: Creating the LeNet5 network with `Sequential` model

Notice that you are just passing in an array of the layers you want your model to contain into the Sequential model constructor. Looking at the `model.summary()`, you can see the model's architecture.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 6)         456       
                                                                 
 max_pooling2d (MaxPooling2D)  (None, 16, 16, 6)       0         
                                                                 
 conv2d_1 (Conv2D)           (None, 16, 16, 16)        2416      
                                                                 
 max_pooling2d_1 (MaxPooling2D)  (None, 8, 8, 16)      0         
                                                                 
 conv2d_2 (Conv2D)           (None, 8, 8, 120)         48120     
 flatten (Flatten)           (None, 7680)          0

 dense (Dense)               (None, 84)            645204

 dense_1 (Dense)             (None, 10)            850

=================================================================
Total params: 697,046
Trainable params: 697,046
Non-trainable params: 0
_________________________________________________________________
```

Output 12.1: LetNet5 network

And just to test out the model, let's go ahead and load the CIFAR-10 dataset and run `model.compile()` and `model.fit`:

```python
from tensorflow as tf

(trainX, trainY), (testX, testY) = tf.keras.datasets.cifar10.load_data()
model.compile(optimizer="adam",
              loss=tf.keras.losses.SparseCategoricalCrossentropy(), metrics="acc")

history = model.fit(x=trainX, y=trainY, batch_size=256, epochs=10,
                    validation_data=(testX, testY))
```

Listing 12.3: Loading CIFAR-10 dataset and train our network

This gives us this output:

```
Epoch 1/10
196/196 [==============================] - 13s 10ms/step - loss: 2.7669 - acc: 0.3648 - val_loss: 1.
Epoch 2/10
196/196 [==============================] - 2s 8ms/step - loss: 1.3883 - acc: 0.5097 - val_loss: 1.36
Epoch 3/10
196/196 [==============================] - 2s 8ms/step - loss: 1.2239 - acc: 0.5694 - val_loss: 1.29
Epoch 4/10
196/196 [==============================] - 2s 8ms/step - loss: 1.1020 - acc: 0.6120 - val_loss: 1.26
Epoch 5/10
196/196 [==============================] - 2s 8ms/step - loss: 0.9931 - acc: 0.6498 - val_loss: 1.28
Epoch 6/10
196/196 [==============================] - 2s 9ms/step - loss: 0.8888 - acc: 0.6903 - val_loss: 1.31
Epoch 7/10
196/196 [==============================] - 2s 8ms/step - loss: 0.7882 - acc: 0.7229 - val_loss: 1.42
Epoch 8/10
196/196 [==============================] - 2s 8ms/step - loss: 0.6915 - acc: 0.7582 - val_loss: 1.45
Epoch 9/10
196/196 [==============================] - 2s 8ms/step - loss: 0.5934 - acc: 0.7931 - val_loss: 1.53
Epoch 10/10
196/196 [==============================] - 2s 8ms/step - loss: 0.5113 - acc: 0.8214 - val_loss: 1.63
```

Output 12.2: Progress during the training of our network

That's pretty good for a first pass at a model. Putting the code for LeNet5 using a `Sequential` model together, you have:

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Input, Flatten, Conv2D, MaxPool2D

(trainX, trainY), (testX, testY) = tf.keras.datasets.cifar10.load_data()

model = Sequential([
          Input(shape=(32,32,3,)),
          Conv2D(6, (5,5), padding="same", activation="relu"),
          MaxPool2D(pool_size=(2,2)),
          Conv2D(16, (5,5), padding="same", activation="relu"),
          MaxPool2D(pool_size=(2, 2)),
          Conv2D(120, (5,5), padding="same", activation="relu"),
          Flatten(),
          Dense(units=84, activation="relu"),
          Dense(units=10, activation="softmax"),
      ])

model.summary()

model.compile(optimizer="adam",
              loss=tf.keras.losses.SparseCategoricalCrossentropy(), metrics="acc")
history = model.fit(x=trainX, y=trainY, batch_size=256, epochs=10,
                    validation_data=(testX, testY))
```

Listing 12.4: Complete example of building a network with `Sequential` and training

Now, let's explore what the other ways of constructing Keras models can do, starting with the functional interface!

12.2 Using Keras' Functional Interface

The next method of constructing Keras models you will explore uses Keras' functional interface. The functional interface uses the layers as functions instead, taking in a Tensor and outputting a Tensor as well. The functional interface is a more flexible way of representing a Keras model as you are not restricted only to sequential models with layers stacked on top of one another. Instead, you can build models that branch into multiple paths, have multiple inputs, etc.

Consider an `Add` layer that takes inputs from two or more paths and adds the tensors together.

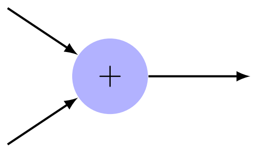


Figure 12.1: Add layer with two inputs

Since this cannot be represented as a linear stack of layers due to the multiple inputs, you are unable to define it using a `Sequential` object. Here's where Keras' functional interface comes in. You can define an `Add` layer with two input tensors as such:

```
from tensorflow.keras.layers import Add
add_layer = Add()([layer1, layer2])
```

Listing 12.5: Using `Add` layer to combine two other layers

Now that you've seen a quick example of the functional interface, let's take a look at what the LeNet5 model that you defined by instantiating a `Sequential` class would look like using a functional interface.

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Input, Flatten, Conv2D, MaxPool2D
from tensorflow.keras.models import Model

input_layer = Input(shape=(32,32,3,))
x = Conv2D(6, (5,5), padding="same", activation="relu")(input_layer)
x = MaxPool2D(pool_size=(2,2))(x)
x = Conv2D(16, (5,5), padding="same", activation="relu")(x)
x = MaxPool2D(pool_size=(2, 2))(x)
x = Conv2D(120, (5,5), padding="same", activation="relu")(x)
x = Flatten()(x)
x = Dense(units=84, activation="relu")(x)
x = Dense(units=10, activation="softmax")(x)

model = Model(inputs=input_layer, outputs=x)

model.summary()
```

Listing 12.6: Creating a LeNet5 network with functional interface

And looking at the model summary:

```
Model: "model"
_________________________________________________________________________
 Layer (type)                Output Shape              Param #
=========================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 6)         456

 max_pooling2d (MaxPooling2D)  (None, 16, 16, 6)       0

 conv2d_1 (Conv2D)           (None, 16, 16, 16)        2416

 max_pooling2d_1 (MaxPooling2D)  (None, 8, 8, 16)      0

 conv2d_2 (Conv2D)           (None, 8, 8, 120)         48120

 flatten (Flatten)           (None, 7680)              0

 dense (Dense)               (None, 84)                645204

 dense_1 (Dense)             (None, 10)                850

=========================================================================
Total params: 697,046
```

```
Trainable params: 697,046
Non-trainable params: 0
_________________________________________________________________
```

Output 12.3: LeNet5 model as creating by functional interface

As you can see, the model architecture is the same for both LeNet5 models you implemented using the functional interface or the `Sequential` class.

Now that you've seen how to use Keras' functional interface, let's look at a model architecture that you can implement using the functional interface but not with the `Sequential` class. For this example, look at the residual block introduced in ResNet. Visually, the residual block looks like this:

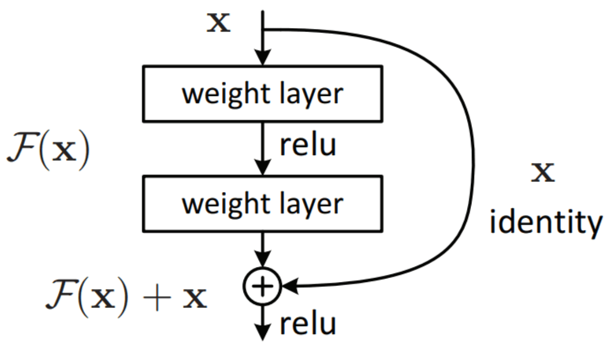

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 12.2: Residual block as used in ResNet

You can see that a model defined using the `Sequential` class would be unable to construct such a block due to the skip connection, which prevents this block from being represented as a simple stack of layers. Using the functional interface this is one way you can define a ResNet block:

```python
def residual_block(x, filters):
    # store the input tensor to be added later as the identity
    identity = x
    x = Conv2D(filters, (3, 3), strides = (1, 1), padding="same")(x)
    x = BatchNormalization()(x)
    x = relu(x)
    x = Conv2D(filters, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Add()([identity, x])
    x = relu(x)

    return x
```

Listing 12.7: Residual block in Keras

Then, you can build a simple network using these residual blocks using the functional interface:

```python
input_layer = Input(shape=(32,32,3,))
x = Conv2D(32, (3, 3), padding="same", activation="relu")(input_layer)
x = residual_block(x, 32)
x = Conv2D(64, (3, 3), strides=(2, 2), padding="same",
           activation="relu")(x)
x = residual_block(x, 64)
x = Conv2D(128, (3, 3), strides=(2, 2), padding="same",
           activation="relu")(x)
x = residual_block(x, 128)
x = Flatten()(x)
x = Dense(units=84, activation="relu")(x)
x = Dense(units=10, activation="softmax")(x)

model = Model(inputs=input_layer, outputs = x)
model.summary()

model.compile(optimizer="adam", loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics="acc")

history = model.fit(x=trainX, y=trainY, batch_size=256, epochs=10,
                    validation_data=(testX, testY))
```

Listing 12.8: A neural network with residual block

Running this code and looking at the model summary and training results:

```
Model: "model"
_____________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
=============================================================================================
 input_1 (InputLayer)           [(None, 32, 32, 3)]  0           []

 conv2d (Conv2D)                (None, 32, 32, 32)   896         ['input_1[0][0]']

 conv2d_1 (Conv2D)              (None, 32, 32, 32)   9248        ['conv2d[0][0]']

 batch_normalization (BatchNorma  (None, 32, 32, 32)  128         ['conv2d_1[0][0]']
 lization)

 tf.nn.relu (TFOpLambda)        (None, 32, 32, 32)   0           ['batch_normalization[0][0]']

 conv2d_2 (Conv2D)              (None, 32, 32, 32)   9248        ['tf.nn.relu[0][0]']

 batch_normalization_1 (BatchNor  (None, 32, 32, 32)  128         ['conv2d_2[0][0]']
 malization)

 add (Add)                      (None, 32, 32, 32)   0           ['conv2d[0][0]',
                                                                  'batch_normalization_1[0][0]']

 tf.nn.relu_1 (TFOpLambda)      (None, 32, 32, 32)   0           ['add[0][0]']

 conv2d_3 (Conv2D)              (None, 16, 16, 64)   18496       ['tf.nn.relu_1[0][0]']

 conv2d_4 (Conv2D)              (None, 16, 16, 64)   36928       ['conv2d_3[0][0]']

 batch_normalization_2 (BatchNor  (None, 16, 16, 64)  256         ['conv2d_4[0][0]']
 malization)

 tf.nn.relu_2 (TFOpLambda)      (None, 16, 16, 64)   0           ['batch_normalization_2[0][0]']
 conv2d_5 (Conv2D)              (None, 16, 16, 64)    36928       ['tf.nn.relu_2[0][0]']

 batch_normalization_3 (BatchNor  (None, 16, 16, 64)   256         ['conv2d_5[0][0]']
 malization)

 add_1 (Add)                    (None, 16, 16, 64)    0           ['conv2d_3[0][0]',
                                                                   'batch_normalization_3[0][0]']

 tf.nn.relu_3 (TFOpLambda)      (None, 16, 16, 64)    0           ['add_1[0][0]']

 conv2d_6 (Conv2D)              (None, 8, 8, 128)     73856       ['tf.nn.relu_3[0][0]']

 conv2d_7 (Conv2D)              (None, 8, 8, 128)     147584      ['conv2d_6[0][0]']

 batch_normalization_4 (BatchNor  (None, 8, 8, 128)    512         ['conv2d_7[0][0]']
 malization)

 tf.nn.relu_4 (TFOpLambda)      (None, 8, 8, 128)     0           ['batch_normalization_4[0][0]']

 conv2d_8 (Conv2D)              (None, 8, 8, 128)     147584      ['tf.nn.relu_4[0][0]']

 batch_normalization_5 (BatchNor  (None, 8, 8, 128)    512         ['conv2d_8[0][0]']
 malization)

 add_2 (Add)                    (None, 8, 8, 128)     0           ['conv2d_6[0][0]',
                                                                   'batch_normalization_5[0][0]']

 tf.nn.relu_5 (TFOpLambda)      (None, 8, 8, 128)     0           ['add_2[0][0]']

 flatten (Flatten)              (None, 8192)          0           ['tf.nn.relu_5[0][0]']

 dense (Dense)                  (None, 84)            688212      ['flatten[0][0]']

 dense_1 (Dense)                (None, 10)            850         ['dense[0][0]']

====================================================================================================
Total params: 1,171,622
Trainable params: 1,170,726
Non-trainable params: 896
____________________________________________________________________________________________________
Epoch 1/10
196/196 [==============================] - 21s 46ms/step - loss: 3.4463 - acc: 0.3635 - val_loss: 1.
Epoch 2/10
196/196 [==============================] - 8s 43ms/step - loss: 1.3267 - acc: 0.5200 - val_loss: 1.3
Epoch 3/10
196/196 [==============================] - 8s 43ms/step - loss: 1.1095 - acc: 0.6062 - val_loss: 1.2
Epoch 4/10
196/196 [==============================] - 9s 44ms/step - loss: 0.9618 - acc: 0.6585 - val_loss: 1.5
Epoch 5/10
196/196 [==============================] - 9s 44ms/step - loss: 0.8656 - acc: 0.6968 - val_loss: 1.1
Epoch 6/10
196/196 [==============================] - 8s 43ms/step - loss: 0.7622 - acc: 0.7361 - val_loss: 1.1
Epoch 7/10
196/196 [==============================] - 9s 44ms/step - loss: 0.6801 - acc: 0.7602 - val_loss: 1.1
Epoch 8/10
```

```
196/196 [==============================] - 8s 43ms/step - loss: 0.6106 - acc: 0.7905 - val_loss: 1.1
Epoch 9/10
196/196 [==============================] - 9s 43ms/step - loss: 0.5367 - acc: 0.8146 - val_loss: 1.2
Epoch 10/10
196/196 [==============================] - 9s 47ms/step - loss: 0.4776 - acc: 0.8348 - val_loss: 1.0
```

Output 12.4: A neural network with residual block

And combining the code for our simple network using residual blocks:

```python
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, BatchNormalization, Add, \
                                    MaxPool2D, Flatten, Dense
from tensorflow.keras.activations import relu
from tensorflow.keras.models import Model

def residual_block(x, filters):
# store the input tensor to be added later as the identity
    identity = x
    x = Conv2D(filters = filters, kernel_size=(3, 3), strides = (1, 1), padding="same")(x)
    x = BatchNormalization()(x)
    x = relu(x)
    x = Conv2D(filters = filters, kernel_size=(3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Add()([identity, x])
    x = relu(x)

    return x

(trainX, trainY), (testX, testY) = tf.keras.datasets.cifar10.load_data()

input_layer = Input(shape=(32,32,3,))
x = Conv2D(32, (3, 3), padding="same", activation="relu")(input_layer)
x = residual_block(x, 32)
x = Conv2D(64, (3, 3), strides=(2, 2), padding="same", activation="relu")(x)
x = residual_block(x, 64)
x = Conv2D(128, (3, 3), strides=(2, 2), padding="same", activation="relu")(x)
x = residual_block(x, 128)
x = Flatten()(x)
x = Dense(units=84, activation="relu")(x)
x = Dense(units=10, activation="softmax")(x)

model = Model(inputs=input_layer, outputs = x)
model.summary()

model.compile(optimizer="adam", loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics="acc")

history = model.fit(x=trainX, y=trainY, batch_size=256, epochs=10,
                    validation_data=(testX, testY))
```

Listing 12.9: Complete code to build a neural network with residual block using functional interface

12.3 Subclassing keras.Model

Keras also provides an object-oriented approach to creating models, which helps with reusability and allows you to represent the models you want to create as classes. This representation might be more intuitive since you can think about models as a set of layers strung together to form your network.

To begin subclassing `keras.Model`, you first need to import it:

```
from tensorflow.keras.models import Model
```

Listing 12.10: Import statement for subclassing

Then, you can start subclassing `Model`. First, you need to build the layers that you want to use in your method calls since you only want to instantiate these layers once instead of each time you call your model. To keep in line with previous examples, let's build a LeNet5 model here as well.

```
class LeNet5(tf.keras.Model):
  def __init__(self):
    super(LeNet5, self).__init__()
    #creating layers in initializer
    self.conv1 = Conv2D(6, (5,5), padding="same", activation="relu")
    self.max_pool2x2 = MaxPool2D(pool_size=(2,2))
    self.conv2 = Conv2D(16, (5,5), padding="same", activation="relu")
    self.conv3 = Conv2D(120, (5,5), padding="same", activation="relu")
    self.flatten = Flatten()
    self.fc2 = Dense(units=84, activation="relu")
    self.fc3 = Dense(units=10, activation="softmax")
```

Listing 12.11: Building a network using subclassing

Then, override the `call` method to define what happens when the model is called. You override it with your model, which uses the layers that you have built in the initializer.

```
def call(self, input_tensor):
  # don't create layers here, need to create the layers in initializer,
  # otherwise you will get the tf.Variable can only be created once error
  conv1 = self.conv1(input_tensor)
  maxpool1 = self.max_pool2x2(conv1)
  conv2 = self.conv2(maxpool1)
  maxpool2 = self.max_pool2x2(conv2)
  conv3 = self.conv3(maxpool2)
  flatten = self.flatten(conv3)
  fc2 = self.fc2(flatten)
  fc3 = self.fc3(fc2)

  return fc3
```

Listing 12.12: The `call()` method added to the subclassed model

It is important to have all the layers created at the class constructor, not inside the `call()` method. This is because the `call()` method will be invoked multiple times with different input tensors. But you want to use the same layer objects in each call to optimize their weight. You can then instantiate your new LeNet5 class and use it as part of a model:

```
input_layer = Input(shape=(32,32,3,))
x = LeNet5()(input_layer)

model = Model(inputs=input_layer, outputs=x)

model.summary(expand_nested=True)
```

Listing 12.13: Creating a LeNet5 network using subclassing

And you can see that the model has the same number of parameters as the previous two versions of LeNet5 that were built previously and has the same structure within it:

```
Model: "model"
_________________________________________________________________________
 Layer (type)                   Output Shape              Param #
=========================================================================
 input_1 (InputLayer)           [(None, 32, 32, 3)]       0

 le_net5 (LeNet5)               (None, 10)                697046
|¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯|
| conv2d (Conv2D)              multiple                  456          |
|                                                                     |
| max_pooling2d (MaxPooling2D)  multiple                 0            |
|                                                                     |
| conv2d_1 (Conv2D)            multiple                  2416         |
|                                                                     |
| conv2d_2 (Conv2D)            multiple                  48120        |
|                                                                     |
| flatten (Flatten)            multiple                  0            |
|                                                                     |
| dense (Dense)                multiple                  645204       |
|                                                                     |
| dense_1 (Dense)              multiple                  850          |
-------------------------------------------------------------------------

=========================================================================
Total params: 697,046
Trainable params: 697,046
Non-trainable params: 0
_________________________________________________________________________
```

Output 12.5: LeNet5 network as created with subclassing

Combining all the code to create your LeNet5 subclass of `keras.Model`:

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Input, Flatten, Conv2D, MaxPool2D
from tensorflow.keras.models import Model

class LeNet5(tf.keras.Model):
  def __init__(self):
    super(LeNet5, self).__init__()
    #creating layers in initializer
    self.conv1 = Conv2D(6, (5,5), padding="same", activation="relu")
    self.max_pool2x2 = MaxPool2D(pool_size=(2,2))
    self.conv2 = Conv2D(16, (5,5), padding="same", activation="relu")
    self.conv3 = Conv2D(120, (5,5), padding="same", activation="relu")
    self.flatten = Flatten()
    self.fc2 = Dense(units=84, activation="relu")
    self.fc3=Dense(units=10, activation="softmax")

  def call(self, input_tensor):
    # don't add layers here, need to create the layers in initializer,
    # otherwise you will get the tf.Variable can only be created once error
    x = self.conv1(input_tensor)
    x = self.max_pool2x2(x)
    x = self.conv2(x)
    x = self.max_pool2x2(x)
    x = self.conv3(x)
    x = self.flatten(x)
    x = self.fc2(x)
    x = self.fc3(x)
    return x

input_layer = Input(shape=(32,32,3,))
x = LeNet5()(input_layer)
model = Model(inputs=input_layer, outputs=x)
model.summary(expand_nested=True)
```

Listing 12.14: Example of creating a neural network using subclassing

12.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Online Resources

The Sequential class. Keras API Reference.
https://keras.io/api/models/sequential/

The Functional API. Keras Developer Guides.
https://keras.io/guides/functional_api/

Making new layers and models via subclassing. Keras Developer Guides.
https://keras.io/guides/making_new_layers_and_models_via_subclassing/

Papers

Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.
https://arxiv.org/pdf/1512.03385.pdf
(The ResNet paper).

12.5 Summary

In this chapter, you have seen three different ways to create models in Keras. In particular, this includes using the `Sequential` class, functional interface, and subclassing `keras.Model`. You have also seen examples of the same LeNet5 model being built using the different methods and a use case that can be done using the functional interface but not with the `Sequential` class.

Specifically, you learned:

- ▷ Different ways that Keras offers to build models
- ▷ How to use the `Sequential` class, functional interface, and subclassing `keras.Model` to build Keras models
- ▷ When to use the different methods to create Keras models

In the next part, you will see more details on how to fine tune a Keras model.

III Advanced Multilayer Perceptrons and Keras

Use Keras Deep Learning Models with scikit-learn

13

Keras is one of the most popular deep learning libraries in Python for research and development because of its simplicity and ease of use. The scikit-learn library is the most popular library for general machine learning in Python. In this chapter, you will discover how to use deep learning models from Keras with the scikit-learn library in Python. This will allow you to leverage the power of the scikit-learn library for tasks like model evaluation and model hyper-parameter optimization. After completing this lesson you will know:

- ▷ How to wrap a Keras model for use with the scikit-learn machine learning library
- ▷ How to easily evaluate Keras models using cross-validation in scikit-learn
- ▷ How to tune Keras model hyperparameters using grid search in scikit-learn

Let's get started.

Overview

This chapter is in three parts; they are:

- ▷ Overview of SciKeras
- ▷ Evaluate Deep Learning Models with Cross-Validation
- ▷ Grid Search Deep Learning Model Parameters

13.1 Overview of SciKeras

Keras is a popular library for deep learning in Python, but the focus of the library is *deep learning*, not all of machine learning. In fact, it strives for minimalism, focusing on only what you need to quickly and simply define and build deep learning models. The scikit-learn library in Python is built upon the SciPy stack for efficient numerical computation. It is a fully featured library for general purpose machine learning and provides many useful utilities in developing deep learning models. Not least of which are:

- ▷ Evaluation of models using resampling methods like k-fold cross-validation
- ▷ Efficient search and evaluation of model hyperparameters

There was a wrapper in the TensorFlow/Keras library to make deep learning models used as classification or regression estimators in scikit-learn. But recently, this wrapper was taken out to become a standalone Python module.

In the following sections, you will work through examples of using the `KerasClassifier` wrapper for a classification neural network created in Keras and used in the scikit-learn library. The test problem is the Pima Indians onset of diabetes classification dataset (see Section 7.1). This is a small dataset with all numerical attributes that is easy to work with.

The following examples assume you have successfully installed TensorFlow 2.x, SciKeras, and scikit-learn. If you use the `pip` for your Python modules, you may install them with:

```
pip install tensorflow scikeras scikit-learn
```

Listing 13.1: Installing required libraries

13.2 Evaluate Deep Learning Models with Cross-Validation

The `KerasClassifier` and `KerasRegressor` classes in SciKeras take an argument `model` which is the name of the function to call to get your model. You must define a function that defines your model, compiles it, and returns it. In the example below, you will define a function `create_model()` that creates a simple multilayer neural network for the problem.

You pass this function name to the `KerasClassifier` class by the `model` argument. You also pass in additional arguments of `epoch=150` and `batch_size=10`. These are automatically bundled up and passed on to the `fit()` function, which is called internally by the `KerasClassifier` class. In this example, you will use the scikit-learn `StratifiedKFold` to perform 10-fold stratified cross-validation. This is a resampling technique that can provide a robust estimate of the performance of a machine learning model on unseen data. Next, use the scikit-learn function `cross_val_score()` to evaluate your model using cross-validation and print the results.

```
# MLP for Pima Indians Dataset with 10-fold cross validation via sklearn
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
import numpy as np

# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

```
# fix random seed for reproducibility
seed = 7
np.random.seed(seed)
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, epochs=150, batch_size=10, verbose=0)
# evaluate using 10-fold cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 13.2: Evaluate a neural network using scikit-learn

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example displays the skill of the model for each epoch. A total of 10 models are created and evaluated, and the final average accuracy is displayed.

```
0.646838691487
```

Output 13.1: The final average accuracy evaluated

You can see that when the Keras model is wrapped that estimating model accuracy can be greatly streamlined, compared to the manual enumeration of cross-validation folds performed in the previous chapter.

In comparison, the following is an equivalent implementation with a neural network model in scikit-learn:

```
# MLP for Pima Indians Dataset with 10-fold cross validation via sklearn
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.neural_network import MLPClassifier
import numpy as np

# fix random seed for reproducibility
seed = 7
np.random.seed(seed)
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = MLPClassifier(hidden_layer_sizes=(12,8), activation='relu',
                        max_iter=150, batch_size=10, verbose=False)
# evaluate using 10-fold cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 13.3: Equivalent implementation in scikit-learn

The role of the `KerasClassifier` is to work as an adapter to make the Keras model work like a `MLPClassifier` object from scikit-learn.

13.3 Grid Search Deep Learning Model Parameters

The previous example showed how easy it is to wrap your deep learning model from Keras and use it in functions from the scikit-learn library. In this example, you will go a step further. The function that you specify to the `model` argument when creating the `KerasClassifier` wrapper can take arguments. You can use these arguments to further customize the construction of the model. In addition, you know you can provide arguments to the `fit()` function.

In this example, you will use grid search to evaluate different configurations for your neural network model and report on the combination that provides the best estimated performance. The `create_model()` function is defined to take two arguments, `optimizer` and `init`, both of which must have default values. This will allow you to evaluate the effect of using different optimization algorithms and weight initialization schemes for your network. After creating your model, you define the arrays of values for the parameter you wish to search, specifically:

- ▷ Optimizers for searching different weight values
- ▷ Initializers for preparing the network weights using different schemes
- ▷ Number of epochs for training the model for different number of exposures to the training dataset
- ▷ Batches for varying the number of samples before weight updates

The options are specified into a dictionary and passed to the configuration of the `GridSearchCV` scikit-learn class. This class will evaluate a version of your neural network model for each combination of parameters ($2 \times 3 \times 3 \times 3$ for the combinations of optimizers, initializations, epochs, and batches). Each combination is then evaluated using the default of 3-fold stratified cross-validation.

That is a lot of models and a lot of computation. This is not a scheme you want to use lightly because it takes a long time to compute. It may be useful for you to design small models with a smaller subset of data that will complete in a reasonable time. It is the case of this experiment (less than 1000 instances and nine attributes). Finally, the performance and combination of configurations for the best model are displayed, followed by the performance of all combinations of parameters.

```
# MLP for Pima Indians Dataset with grid search via sklearn
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import GridSearchCV
import numpy as np

# Function to create model, required for KerasClassifier
def create_model(optimizer='rmsprop', init='glorot_uniform'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), kernel_initializer=init, activation='relu'))
    model.add(Dense(8, kernel_initializer=init, activation='relu'))
    model.add(Dense(1, kernel_initializer=init, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

# fix random seed for reproducibility
seed = 7
np.random.seed(seed)
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, verbose=0)
print(model.get_params().keys())
# grid search epochs, batch size and optimizer
optimizers = ['rmsprop', 'adam']
init = ['glorot_uniform', 'normal', 'uniform']
epochs = [50, 100, 150]
sizes = [5, 10, 20]
param_grid = dict(optimizer=optimizers, epochs=epochs, batch_size=sizes, model__init=init)
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 13.4: Grid Search Neural Network Parameters Using scikit-learn

Note that in the dictionary `param_grid`, `model__init` was used as the key for the `init` argument to our `create_model()` function. The prefix `model__` is required for `KerasClassifier` models in SciKeras to provide custom arguments.

This might take about 5 minutes to complete on your workstation executed on the CPU (rather than GPU). Running the example shows the results below. You can see that the grid search discovered that using a uniform initialization scheme, rmsprop optimizer, 150 epochs, and a batch size of 10 achieved the best cross-validation score of approximately 77% on this problem.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Best: 0.772167 using {'batch_size': 10, 'epochs': 150, 'model__init': 'normal', 'optimizer': 'rmspro
0.691359 (0.024495) with: {'batch_size': 5, 'epochs': 50, 'model__init': 'glorot_uniform', 'optimize
0.690094 (0.026691) with: {'batch_size': 5, 'epochs': 50, 'model__init': 'glorot_uniform', 'optimize
0.704482 (0.036013) with: {'batch_size': 5, 'epochs': 50, 'model__init': 'normal', 'optimizer': 'rms
0.727884 (0.016890) with: {'batch_size': 5, 'epochs': 50, 'model__init': 'normal', 'optimizer': 'ada
0.709600 (0.030281) with: {'batch_size': 5, 'epochs': 50, 'model__init': 'uniform', 'optimizer': 'rm
...
```

Output 13.2: Output of grid search neural network parameters using scikit-learn

13.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Online Resources

SciKeras documentation.
https://www.adriangb.com/scikeras/stable/

Stratified K-Folds cross-validator. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

Grid search cross-validator. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

13.5 Summary

In this chapter, you discovered how to wrap your Keras deep learning models and use them in the scikit-learn general machine learning library. You learned:

- ▷ Specifically how to wrap Keras models so that they can be used with the scikit-learn machine learning library.
- ▷ How to use a wrapped Keras model as part of evaluating model performance in scikit-learn.
- ▷ How to perform hyperparameter tuning in scikit-learn using a wrapped Keras model.

You can see that using scikit-learn for standard machine learning operations such as model evaluation and model hyperparameter optimization can save a lot of time over implementing these schemes yourself. Wrapping your model allowed you to leverage powerful tools from scikit-learn to fit your deep learning models into your general machine learning process. We will see more examples in the next chapter.

14 How to Grid Search Hyperparameters for Deep Learning Models

Hyperparameters are the parameters of a neural network that is fixed by design and not tuned by training. Examples are the number of hidden layers and the choice of activation functions. Hyperparameter optimization is a big part of deep learning. The reason is that neural networks are notoriously difficult to configure, and a lot of parameters need to be set. On top of that, individual models can be very slow to train.

In this chapter, you will discover how to use the grid search capability from the scikit-learn Python machine learning library to tune the hyperparameters of Keras deep learning models. After reading this chapter, you will know:

- ▷ How to wrap Keras models for use in scikit-learn and how to use grid search
- ▷ How to grid search common neural network parameters, such as learning rate, dropout rate, epochs, and number of neurons
- ▷ How to define your own hyperparameter tuning experiments on your own projects

Let's get started.

Overview

In this chapter, I want to show you both how you can use the scikit-learn grid search capability and give you a suite of examples that you can copy-and-paste into your own project as a starting point. Below is a list of the topics we are going to cover:

- ▷ How to use Keras models in scikit-learn
- ▷ How to use grid search in scikit-learn
- ▷ How to tune batch size and training epochs
- ▷ How to tune optimization algorithms
- ▷ How to tune learning rate and momentum
- ▷ How to tune network weight initialization
- ▷ How to tune activation functions
- ▷ How to tune dropout regularization

▷ How to tune the number of neurons in the hidden layer

14.1 How to Use Keras Models in scikit-learn

Keras models can be used in scikit-learn by wrapping them with the `KerasClassifier` or `KerasRegressor` class from the module SciKeras. You may need to run the command in Listing 13.1 to install the module.

To use these wrappers, you must define a function that creates and returns your Keras sequential model, then pass this function to the `model` argument when constructing the `KerasClassifier` class. For example:

```python
def create_model():
    ...
    return model

model = KerasClassifier(model=create_model)
```

Listing 14.1: Using KerasClassifier wrapper

The constructor for the `KerasClassifier` class can take default arguments that are passed on to the calls to `model.fit()`, such as the number of epochs and the batch size. For example:

```python
def create_model():
    ...
    return model

model = KerasClassifier(model=create_model, epochs=10)
```

Listing 14.2: Using KerasClassifier with epochs argument

The constructor for the `KerasClassifier` class can also take new arguments that can be passed to your custom `create_model()` function. These new arguments must also be defined in the signature of your `create_model()` function with default parameters. For example:

```python
def create_model(dropout_rate=0.0):
    ...
    return model

model = KerasClassifier(model=create_model, dropout_rate=0.2)
```

Listing 14.3: Using KerasClassifier with dropout rate argument

You can learn more about these from the SciKeras documentation.

14.2 How to Use Grid Search in scikit-learn

Grid search is a model hyperparameter optimization technique. It simply exhaust all combinations of the hyperparameters and find the one that gave the best score. In scikit-learn, this technique is provided in the `GridSearchCV` class. When constructing this class, you must provide a dictionary of hyperparameters to evaluate in the `param_grid` argument. This is a map of the model parameter name and an array of values to try.

By default, accuracy is the score that is optimized, but other scores can be specified in the `score` argument of the `GridSearchCV` constructor. By default, the grid search will only use one thread. By setting the `n_jobs` argument in the `GridSearchCV` constructor to -1, the process will use all cores on your machine. However, sometimes this may interfere with the main neural network training process. The `GridSearchCV` process will then construct and evaluate one model for each combination of parameters. Cross-validation is used to evaluate each individual model, and the default of 3-fold cross-validation is used, although you can override this by specifying the `cv` argument to the `GridSearchCV` constructor.

Below is an example of defining a simple grid search:

```
param_grid = dict(epochs=[10,20,30])
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
```

Listing 14.4: Running grid search using scikit-learn

Once completed, you can access the outcome of the grid search in the result object returned from `grid.fit()`. The `best_score_` member provides access to the best score observed during the optimization procedure, and the `best_params_` describes the combination of parameters that achieved the best results. You can learn more about the `GridSearchCV` class in the scikit-learn API documentation.

14.3 Problem Description

Now that you know how to use Keras models with scikit-learn and how to use grid search in scikit-learn, let's look at a bunch of examples.

All examples will be demonstrated on a small standard machine learning dataset called the Pima Indians onset of diabetes classification dataset. This is a small dataset with all numerical attributes that is easy to work with. (See Section 7.1) As you proceed through the examples in this chapter, you will aggregate the best parameters. This is not the best way to grid search because parameters can interact, but it is good for demonstration purposes.

Note on Parallelizing Grid Search

All examples are configured to use parallelism (`n_jobs=-1`). If you get an error like the one below:

```
INFO (theano.gof.compilelock): Waiting for existing lock by process '55614' (I am process '55613')
INFO (theano.gof.compilelock): To manually release the lock, delete ...
```

Output 14.1: Error message due to parallelism

Kill the process and change the code to not perform the grid search in parallel; set `n_jobs=1`.

14.4 How to Tune Batch Size and Number of Epochs

In this first simple example, you will look at tuning the batch size and number of epochs used when fitting the network.

The batch size in iterative gradient descent is the number of patterns shown to the network before the weights are updated. It is also an optimization in the training of the network, defining how many patterns to read at a time and keep in memory. The number of epochs is the number of times the entire training dataset is shown to the network during training. Some networks are sensitive to the batch size, such as LSTM recurrent neural networks and Convolutional Neural Networks. Here you will evaluate a suite of different minibatch sizes from 10 to 100 in steps of 20.

The full code listing is provided below:

```python
# Use scikit-learn to grid search the batch size and epochs
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
tf.random.set_seed(seed)
# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, verbose=0)
# define the grid search parameters
batch_size = [10, 20, 40, 60, 80, 100]
epochs = [10, 50, 100]
param_grid = dict(batch_size=batch_size, epochs=epochs)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
```

```python
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 14.5: Grid search on batch size and number of epochs

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running this example produces the following output:

```
Best: 0.705729 using {'batch_size': 10, 'epochs': 100}
0.597656 (0.030425) with: {'batch_size': 10, 'epochs': 10}
0.686198 (0.017566) with: {'batch_size': 10, 'epochs': 50}
0.705729 (0.017566) with: {'batch_size': 10, 'epochs': 100}
0.494792 (0.009207) with: {'batch_size': 20, 'epochs': 10}
0.675781 (0.017758) with: {'batch_size': 20, 'epochs': 50}
0.683594 (0.011049) with: {'batch_size': 20, 'epochs': 100}
0.535156 (0.053274) with: {'batch_size': 40, 'epochs': 10}
0.622396 (0.009744) with: {'batch_size': 40, 'epochs': 50}
0.671875 (0.019918) with: {'batch_size': 40, 'epochs': 100}
0.592448 (0.042473) with: {'batch_size': 60, 'epochs': 10}
0.660156 (0.041707) with: {'batch_size': 60, 'epochs': 50}
0.674479 (0.006639) with: {'batch_size': 60, 'epochs': 100}
0.476562 (0.099896) with: {'batch_size': 80, 'epochs': 10}
0.608073 (0.033197) with: {'batch_size': 80, 'epochs': 50}
0.660156 (0.011500) with: {'batch_size': 80, 'epochs': 100}
0.615885 (0.015073) with: {'batch_size': 100, 'epochs': 10}
0.617188 (0.039192) with: {'batch_size': 100, 'epochs': 50}
0.632812 (0.019918) with: {'batch_size': 100, 'epochs': 100}
```

Output 14.2: Result of grid search on batch size and number of epochs

You can see that the batch size of 10 and 100 epochs achieved the best result of about 70% accuracy.

14.5 How to Tune the Training Optimization Algorithm

Keras offers a suite of different state-of-the-art optimization algorithms. In this example, you will tune the optimization algorithm used to train the network, each with default parameters. This is an odd example because often, you will choose one approach a priori and instead focus on tuning its parameters on your problem (see the next example). Here, you will evaluate the suite of optimization algorithms supported by the Keras API.

The full code listing is provided below:

```python
# Use scikit-learn to grid search the optimization algorithms
import numpy as np
import tensorflow as tf
```

```python
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # return model without compile
    return model
# fix random seed for reproducibility
seed = 7
tf.random.set_seed(seed)
# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, loss="binary_crossentropy",
                        epochs=100, batch_size=10, verbose=0)
# define the grid search parameters
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']
param_grid = dict(optimizer=optimizer)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 14.6: Grid search on optimization algorithms

Note the function `create_model()` defined above does not return a compiled model like that one in the previous example. This is because setting an optimizer for a Keras model is done in the `compile()` function call; hence it is better to leave it to the `KerasClassifier` wrapper and the `GridSearchCV` model. Also, note that you specified `loss="binary_crossentropy"` in the wrapper as it should also be set during the `compile()` function call.

Running this example produces the following output:

```
Best: 0.697917 using {'optimizer': 'Adam'}
0.674479 (0.033804) with: {'optimizer': 'SGD'}
0.649740 (0.040386) with: {'optimizer': 'RMSprop'}
0.595052 (0.032734) with: {'optimizer': 'Adagrad'}
0.348958 (0.001841) with: {'optimizer': 'Adadelta'}
0.697917 (0.038051) with: {'optimizer': 'Adam'}
0.652344 (0.019918) with: {'optimizer': 'Adamax'}
0.684896 (0.011201) with: {'optimizer': 'Nadam'}
```

Output 14.3: Result of grid search on optimization algorithms

The `KerasClassifier` wrapper will not compile your model again if the model is already compiled. Hence the other way to run `GridSearchCV` is to set the optimizer as an argument to the `create_model()` function, which returns an appropriately compiled model like the following:

```python
# Use scikit-learn to grid search the optimization algorithms
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model(optimizer='adam'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
tf.random.set_seed(seed)
# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, epochs=100, batch_size=10, verbose=0)
# define the grid search parameters
optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']
param_grid = dict(model__optimizer=optimizer)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 14.7: Grid search on optimization algorithms with compiled model

Note that in the above, you have the prefix `model__` in the parameter dictionary `param_grid`. This is required for the `KerasClassifier` in the SciKeras module to make clear that the parameter needs to *route* into the `create_model()` function as arguments, rather than some parameter to set up in `compile()` or `fit()`. See also the routed parameter section of SciKeras documentation.

Running this example produces the following output:

```
Best: 0.697917 using {'model__optimizer': 'Adam'}
0.636719 (0.019401) with: {'model__optimizer': 'SGD'}
0.683594 (0.020915) with: {'model__optimizer': 'RMSprop'}
0.585938 (0.038670) with: {'model__optimizer': 'Adagrad'}
0.518229 (0.120624) with: {'model__optimizer': 'Adadelta'}
0.697917 (0.049445) with: {'model__optimizer': 'Adam'}
0.652344 (0.027805) with: {'model__optimizer': 'Adamax'}
0.686198 (0.012890) with: {'model__optimizer': 'Nadam'}
```

Output 14.4: Result of grid search on optimization algorithms, using compiled model

The results suggest that the ADAM optimization algorithm is the best with a score of about 70% accuracy.

14.6 How to Tune Learning Rate and Momentum

It is common to pre-select an optimization algorithm to train your network and tune its parameters. By far, the most common optimization algorithm is plain old Stochastic Gradient Descent (SGD) because it is so well understood. In this example, you will look at optimizing the SGD learning rate and momentum parameters.

The learning rate controls how much to update the weight at the end of each batch, and the momentum controls how much to let the previous update influence the current weight update. You will try a suite of small standard learning rates and a momentum values from 0.2 to 0.8 in steps of 0.2, as well as 0.9 (because it can be a popular value in practice). In Keras, the way to set the learning rate and momentum is the following:

```
...
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.2)
```

Listing 14.8: Setting learning rate and momentum in SGD

In the SciKeras wrapper, you will *route* the parameters to the optimizer with the prefix `optimizer__`. Generally, it is a good idea to also include the number of epochs in an optimization like this as there is a dependency between the amount of learning per batch (learning rate), the number of updates per epoch (batch size), and the number of epochs.

The full code listing is provided below:

```
# Use scikit-learn to grid search the learning rate and momentum
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from scikeras.wrappers import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    return model
# fix random seed for reproducibility
seed = 7
tf.random.set_seed(seed)
# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, loss="binary_crossentropy",
                        optimizer="SGD", epochs=100, batch_size=10, verbose=0)
# define the grid search parameters
learn_rate = [0.001, 0.01, 0.1, 0.2, 0.3]
momentum = [0.0, 0.2, 0.4, 0.6, 0.8, 0.9]
param_grid = dict(optimizer__learning_rate=learn_rate, optimizer__momentum=momentum)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 14.9: Grid search on learning rate and momentum

Running this example produces the following output.

```
Best: 0.686198 using {'optimizer__learning_rate': 0.001, 'optimizer__momentum': 0.0}
0.686198 (0.036966) with: {'optimizer__learning_rate': 0.001, 'optimizer__momentum': 0.0}
0.651042 (0.009744) with: {'optimizer__learning_rate': 0.001, 'optimizer__momentum': 0.2}
0.652344 (0.038670) with: {'optimizer__learning_rate': 0.001, 'optimizer__momentum': 0.4}
0.656250 (0.065907) with: {'optimizer__learning_rate': 0.001, 'optimizer__momentum': 0.6}
0.671875 (0.022326) with: {'optimizer__learning_rate': 0.001, 'optimizer__momentum': 0.8}
0.661458 (0.015733) with: {'optimizer__learning_rate': 0.001, 'optimizer__momentum': 0.9}
...
```

Output 14.5: Result of grid search on learning rate and momentum

You can see that SGD is not very good on this problem; nevertheless, the best results were achieved using a learning rate of 0.001 and a momentum of 0.0 with an accuracy of about 68%.

14.7 How to Tune Network Weight Initialization

Neural network weight initialization used to be simple: use small random values. Now there is a suite of different techniques to choose from. Keras provides a laundry list. In this example, you will look at tuning the selection of network weight initialization by evaluating all the available techniques.

You will use the same weight initialization method on each layer. Ideally, it may be better to use different weight initialization schemes according to the activation function used on each layer. In the example below, you will use a rectifier for the hidden layer. Use sigmoid for the output layer because the predictions are binary. The weight initialization is now an argument to `create_model()` function, where you need to use the `model__` prefix to ask the `KerasClassifier` to route the parameter to the model creation function.

The full code listing is provided below:

```python
# Use scikit-learn to grid search the weight initialization
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model(init_mode='uniform'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), kernel_initializer=init_mode,
                    activation='relu'))
    model.add(Dense(1, kernel_initializer=init_mode, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
tf.random.set_seed(seed)
# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, epochs=100, batch_size=10, verbose=0)
# define the grid search parameters
init_mode = ['uniform', 'lecun_uniform', 'normal', 'zero', 'glorot_normal',
             'glorot_uniform', 'he_normal', 'he_uniform']
param_grid = dict(model__init_mode=init_mode)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 14.10: Grid search on weight initialization

Running this example produces the following output.

```
Best: 0.716146 using {'model__init_mode': 'uniform'}
0.716146 (0.034987) with: {'model__init_mode': 'uniform'}
0.678385 (0.029635) with: {'model__init_mode': 'lecun_uniform'}
0.716146 (0.030647) with: {'model__init_mode': 'normal'}
0.651042 (0.001841) with: {'model__init_mode': 'zero'}
0.695312 (0.027805) with: {'model__init_mode': 'glorot_normal'}
0.690104 (0.023939) with: {'model__init_mode': 'glorot_uniform'}
0.647135 (0.057880) with: {'model__init_mode': 'he_normal'}
0.665365 (0.026557) with: {'model__init_mode': 'he_uniform'}
```

Output 14.6: Result of grid search on weight initialization

You can see that the best results were achieved with a uniform weight initialization scheme achieving a performance of about 72%.

14.8 How to Tune the Neuron Activation Function

The activation function controls the nonlinearity of individual neurons and when to fire. Generally, the rectifier activation function is the most popular. However, it used to be the sigmoid and the tanh functions, and these functions may still be more suitable for different problems.

In this example, you will evaluate the suite of different activation functions available in Keras. You will only use these functions in the hidden layer, as a sigmoid activation function is required in the output for the binary classification problem. Similar to the previous example, this is an argument to the `create_model()` function, and you will use the `model__` prefix for the `GridSearchCV` parameter grid. Generally, it is a good idea to prepare data to the range of the different transfer functions, which you will not do in this case.

The full code listing is provided below:

```python
# Use scikit-learn to grid search the activation function
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model(activation='relu'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), kernel_initializer='uniform',
                    activation=activation))
    model.add(Dense(1, kernel_initializer='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
tf.random.set_seed(seed)
```

```
# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, epochs=100, batch_size=10, verbose=0)
# define the grid search parameters
activation = ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid',
              'hard_sigmoid', 'linear']
param_grid = dict(model__activation=activation)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 14.11: Grid search on activation function

Running this example produces the following output.

```
Best: 0.710938 using {'model__activation': 'linear'}
0.651042 (0.001841) with: {'model__activation': 'softmax'}
0.703125 (0.012758) with: {'model__activation': 'softplus'}
0.671875 (0.009568) with: {'model__activation': 'softsign'}
0.710938 (0.024080) with: {'model__activation': 'relu'}
0.669271 (0.019225) with: {'model__activation': 'tanh'}
0.675781 (0.011049) with: {'model__activation': 'sigmoid'}
0.677083 (0.004872) with: {'model__activation': 'hard_sigmoid'}
0.710938 (0.034499) with: {'model__activation': 'linear'}
```

Output 14.7: Result of grid search on activation function

Surprisingly (to me at least), the `linear` activation function achieved the best results with an accuracy of about 71%.

14.9 How to Tune Dropout Regularization

In this example, you will look at tuning the dropout rate for regularization in an effort to limit overfitting and improve the model's ability to generalize. For the best results, dropout is best combined with a weight constraint such as the max norm constraint. This involves fitting both the dropout percentage and the weight constraint. We will try dropout percentages between 0.0 and 0.9 (1.0 does not make sense) and MaxNorm weight constraint values between 0 and 5. The full code listing is provided below.

```python
# Use scikit-learn to grid search the dropout rate
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import MaxNorm
from scikeras.wrappers import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model(dropout_rate, weight_constraint):
    # create model
    model = Sequential()
    model.add(Dense(12, input_shape=(8,), kernel_initializer='uniform',
                    activation='linear', kernel_constraint=MaxNorm(weight_constraint)))
    model.add(Dropout(dropout_rate))
    model.add(Dense(1, kernel_initializer='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
seed = 7
tf.random.set_seed(seed)
# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
print(dataset.dtype, dataset.shape)
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, epochs=100, batch_size=10, verbose=0)
# define the grid search parameters
weight_constraint = [1.0, 2.0, 3.0, 4.0, 5.0]
dropout_rate = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
param_grid = dict(model__dropout_rate=dropout_rate,
                  model__weight_constraint=weight_constraint)
#param_grid = dict(model__dropout_rate=dropout_rate)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 14.12: Grid search on dropout rate

Running this example produces the following output.

```
Best: 0.766927 using {'model__dropout_rate': 0.2, 'model__weight_constraint': 3.0}
0.729167 (0.021710) with: {'model__dropout_rate': 0.0, 'model__weight_constraint': 1.0}
0.746094 (0.022326) with: {'model__dropout_rate': 0.0, 'model__weight_constraint': 2.0}
0.753906 (0.022097) with: {'model__dropout_rate': 0.0, 'model__weight_constraint': 3.0}
0.750000 (0.012758) with: {'model__dropout_rate': 0.0, 'model__weight_constraint': 4.0}
0.751302 (0.012890) with: {'model__dropout_rate': 0.0, 'model__weight_constraint': 5.0}
...
```

Output 14.8: Result of grid search on dropout rate

You can see that the dropout rate of 20% and the MaxNorm weight constraint of 3 resulted in the best accuracy of about 77%. You may notice some of the result is `nan`. Probably it is due to the issue that the input is not normalized and you may run into a degenerated model by chance.

14.10 How to Tune the Number of Neurons in the Hidden Layer

The number of neurons in a layer is an important parameter to tune. Generally the number of neurons in a layer controls the representational capacity of the network, at least at that point in the topology. Generally, a large enough single layer network can approximate any other neural network, due to the universal approximation theorem.

In this example, you will look at tuning the number of neurons in a single hidden layer. you will try values from 1 to 30 in steps of 5. A larger network requires more training and at least the batch size and number of epochs should ideally be optimized with the number of neurons.

The full code listing is provided below.

```python
# Use scikit-learn to grid search the number of neurons
import numpy as np
import tensorflow as tf
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from scikeras.wrappers import KerasClassifier
from tensorflow.keras.constraints import MaxNorm
# Function to create model, required for KerasClassifier
def create_model(neurons):
    # create model
    model = Sequential()
    model.add(Dense(neurons, input_shape=(8,), kernel_initializer='uniform',
                    activation='linear', kernel_constraint=MaxNorm(4)))
    model.add(Dropout(0.2))
    model.add(Dense(1, kernel_initializer='uniform', activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# fix random seed for reproducibility
```

```
seed = 7
tf.random.set_seed(seed)
# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(model=create_model, epochs=100, batch_size=10, verbose=0)
# define the grid search parameters
neurons = [1, 5, 10, 15, 20, 25, 30]
param_grid = dict(model__neurons=neurons)
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=3)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Listing 14.13: Grid search on number of neurons in hidden layer

Running this example produces the following output.

```
Best: 0.729167 using {'model__neurons': 30}
0.701823 (0.010253) with: {'model__neurons': 1}
0.717448 (0.011201) with: {'model__neurons': 5}
0.717448 (0.008027) with: {'model__neurons': 10}
0.720052 (0.019488) with: {'model__neurons': 15}
0.709635 (0.004872) with: {'model__neurons': 20}
0.708333 (0.003683) with: {'model__neurons': 25}
0.729167 (0.009744) with: {'model__neurons': 30}
```

Output 14.9: Grid search on number of neurons in hidden layer

You can see that the best results were achieved with a network with 30 neurons in the hidden layer with an accuracy of about 73%.

14.11 Tips for Hyperparameter Optimization

This section lists some handy tips to consider when tuning hyperparameters of your neural network.

- ▷ **k-Fold Cross-Validation**. You can see that the results from the examples in this chapter show some variance. A default cross-validation of 3 was used, but perhaps $k = 5$ or $k = 10$ would be more stable. Carefully choose your cross-validation configuration to ensure your results are stable.
- ▷ **Review the Whole Grid**. Do not just focus on the best result, review the whole grid of results and look for trends to support configuration decisions.

- ▷ **Parallelize**. Use all your cores if you can, neural networks are slow to train and we often want to try a lot of different parameters. Consider spinning up a lot of AWS instances.
- ▷ **Use a Sample of Your Dataset**. Because networks are slow to train, try training them on a smaller sample of your training dataset, just to get an idea of general directions of parameters rather than optimal configurations.
- ▷ **Start with Coarse Grids**. Start with coarse-grained grids and zoom into finer grained grids once you can narrow the scope.
- ▷ **Do Not Transfer Results**. Results are generally problem specific. Try to avoid favorite configurations on each new problem that you see. It is unlikely that optimal results you discover on one problem will transfer to your next project. Instead look for broader trends like number of layers or relationships between parameters.
- ▷ **Reproducibility is a Problem**. Although we set the seed for the random number generator in NumPy, the results are not 100% reproducible. There is more to reproducibility when grid searching wrapped Keras models than is presented in this chapter.

14.12 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Online Resources

Optimizers. Keras API Reference.
https://keras.io/api/optimizers/

Layer Weight Initializers. Keras API Reference.
https://keras.io/api/layers/initializers/

SciKeras documentation.
https://www.adriangb.com/scikeras/stable/

Routed parameters. SciKeras documentation.
https://www.adriangb.com/scikeras/stable/advanced.html#routed-parameters

Stratified K-Folds cross-validator. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

Grid search cross-validator. scikit-learn documentation.
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Universal approximation theorem. Wikipedia.
https://en.wikipedia.org/wiki/Universal_approximation_theorem

14.13 Summary

In this chapter, you discovered how you can tune the hyperparameters of your deep learning networks in Python using Keras and scikit-learn. Specifically, you learned:

- ▷ How to wrap Keras models for use in scikit-learn and how to use grid search.
- ▷ How to grid search a suite of different standard neural network parameters for Keras models.
- ▷ How to design your own hyperparameter optimization experiments.

In the next chapter, you will learn how to preserve a Keras model after training.

15 Save and Load Your Keras Model with Serialization

Since deep learning models can take hours, days, and even weeks to train, it is important to know how to save and load them from disk. In this chapter, you will discover how to save your Keras models to files and load them up again to make predictions. After reading this chapter, you will know:

- ▷ How to save model weights and model architecture in separate files
- ▷ How to save model architecture in both YAML and JSON format
- ▷ How to save model weights and architecture into a single file for later use

Let's get started.

Overview

Keras separates the concerns of saving your model architecture and saving your model weights. Model weights are saved to HDF5 format. This grid format is ideal for storing multi-dimensional arrays of numbers. The model structure can be described and saved using two different formats: JSON and YAML.

In this chapter, you will look at three examples of saving and loading your model to a file:

- ▷ Save Model to JSON
- ▷ Save Model to YAML
- ▷ Save Model to HDF5

The examples will use the same simple network trained on the Pima Indians onset of diabetes binary classification dataset. (See Section 7.1)

Note: Saving models requires that you have the h5py library installed. It is usually installed as a dependency with TensorFlow. You can also install it easily as running:

```
sudo pip install h5py
```

15.1 Save Your Neural Network Model to JSON

JSON is a simple file format for describing data hierarchically. Keras provides the ability to describe any model using JSON format with a `to_json()` function. This can be saved to file and later loaded via the `model_from_json()` function that will create a new model from the JSON specification.

The weights are saved directly from the model using the `save_weights()` function and later loaded using the symmetrical `load_weights()` function. The example below trains and evaluates a simple model on the Pima Indians dataset. The model structure is then converted to JSON format and written to `model.json` in the local directory. The network weights are written to `model.h5` in the local directory.

The model and weight data is loaded from the saved files, and a new model is created. It is important to compile the loaded model before it is used. This is so that predictions made using the model can use the appropriate efficient computation from the Keras backend. The model is evaluated in the same way, printing the same evaluation score.

```python
# MLP for Pima Indians Dataset Serialize to JSON and HDF5
from tensorflow.keras.models import Sequential, model_from_json
from tensorflow.keras.layers import Dense
import numpy
import os
# fix random seed for reproducibility
numpy.random.seed(7)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")

# later...
# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")

# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop',
                     metrics=['accuracy'])
score = loaded_model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

Listing 15.1: Serialize model to JSON format

Running this example provides the output below. It shows first the accuracy of the trained model, the saving of the model to disk in JSON format, the loading of the model and finally the re-evaluation of the loaded model achieving the same accuracy.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
acc: 78.78%
Saved model to disk
Loaded model from disk
acc: 78.78%
```

Output 15.1: Sample output from serializing model to JSON ofrmat

The JSON format of the model looks like the following:

```
{
  "class_name": "Sequential",
  "config": {
    "name": "sequential",
    "layers": [
      {
        "class_name": "InputLayer",
        "config": {
          "batch_input_shape": [
            null,
            8
          ],
          "dtype": "float32",
          "sparse": false,
          "ragged": false,
          "name": "dense_input"
        }
      },
      ...
      {
        "class_name": "Dense",
        "config": {
          "name": "dense_2",
          "trainable": true,
          "dtype": "float32",
          "units": 1,
          "activation": "sigmoid",
          "use_bias": true,
          "kernel_initializer": {
            "class_name": "GlorotUniform",
            "config": {
              "seed": null
            }
          },
          "bias_initializer": {
            "class_name": "Zeros",
            "config": {}
          },
          "kernel_regularizer": null,
          "bias_regularizer": null,
          "activity_regularizer": null,
          "kernel_constraint": null,
          "bias_constraint": null
        }
      }
    ]
  },
  "keras_version": "2.9.0",
  "backend": "tensorflow"
}
```

Listing 15.2: Sample JSON model file

15.2 Save Your Neural Network Model to YAML

Note: This method only applies to TensorFlow 2.5 or earlier. If you run it in later versions of TensorFlow, you will see a RuntimeError with the message “Method `model.to_yaml()` has been removed due to security risk of arbitrary code execution. Please use `model.to_json()` instead.”

This example is much the same as the above JSON example, except the YAML format is used for the model specification. This example assumes that you have PyYAML 5, which can be installed with:

```
sudo pip install PyYAML
```

Listing 15.3: Installing PyYAML module

In this example, the model is described using YAML, saved to file `model.yaml`, and later loaded into a new model via the `model_from_yaml()` function. Weights are handled the same way as above in the HDF5 format as `model.h5`.

```python
# MLP for Pima Indians Dataset serialize to YAML and HDF5
from tensorflow.keras.models import Sequential, model_from_yaml
from tensorflow.keras.layers import Dense
import numpy
import os
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

# serialize model to YAML
model_yaml = model.to_yaml()
with open("model.yaml", "w") as yaml_file:
    yaml_file.write(model_yaml)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")

# later...

# load YAML and create model
yaml_file = open('model.yaml', 'r')
loaded_model_yaml = yaml_file.read()
yaml_file.close()
loaded_model = model_from_yaml(loaded_model_yaml)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")

# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop',
                     metrics=['accuracy'])
```

```
score = loaded_model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100))
```

Listing 15.4: Serialize model to YAML format

Running the example displays the following output.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
acc: 78.78%
Saved model to disk
Loaded model from disk
acc: 78.78%
```

Output 15.2: Sample output from serializing model to YAML format

The model described in YAML format looks like the following:

```
backend: tensorflow
class_name: Sequential
config:
  layers:
  - class_name: Dense
    config:
      activation: relu
      activity_regularizer: null
      batch_input_shape: !!python/tuple
      - null
      - 8
      bias_constraint: null
      bias_initializer:
        class_name: Zeros
        config: {}
      bias_regularizer: null
      dtype: float32
      kernel_constraint: null
      kernel_initializer:
        class_name: VarianceScaling
        config:
          distribution: uniform
          mode: fan_avg
          scale: 1.0
          seed: null
      kernel_regularizer: null
      name: dense_1
      trainable: true
      units: 12
      use_bias: true
...
  - class_name: Dense
```

```
    config:
      activation: sigmoid
      activity_regularizer: null
      bias_constraint: null
      bias_initializer:
        class_name: Zeros
        config: {}
      bias_regularizer: null
      dtype: float32
      kernel_constraint: null
      kernel_initializer:
        class_name: VarianceScaling
        config:
          distribution: uniform
          mode: fan_avg
          scale: 1.0
          seed: null
      kernel_regularizer: null
      name: dense_3
      trainable: true
      units: 1
      use_bias: true
  name: sequential_1
keras_version: 2.2.5
```

Listing 15.5: Sample YAML model file

15.3 Save Model Weights and Architecture Together

Keras also supports a simpler interface to save both the model weights and model architecture together into a single H5 file. Saving the model in this way includes everything you need to know about the model, including:

- ▷ Model weights
- ▷ Model architecture
- ▷ Model compilation details (loss and metrics)
- ▷ Model optimizer state

This means that you can load and use the model directly without having to re-compile it as you did to in the examples above. This is the preferred way for saving and loading your Keras model.

How to Save a Keras Model

You can save your model by calling the `save()` function on the model and specifying the filename. The example below demonstrates this by first fitting a model, evaluating it, and saving it to the file `model.h5`.

```
# MLP for Pima Indians Dataset saved to single file
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# load pima indians dataset
dataset = loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# define model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
# save model and architecture to single file
model.save("model.h5")
print("Saved model to disk")
```

Listing 15.6: Example of saving weights and architecture to single file

Running the example fits the model, summarizes the model's performance on the training dataset, and saves the model to file.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
acc: 77.73%
Saved model to disk
```

Output 15.3: Sample output from saving weights and architecture to a single file

You can later load this model from the file and use it. Note that in the Keras library, there is another function doing the same, as follows:

```
...
# equivalent to: model.save("model.h5")
from tensorflow.keras.models import save_model
save_model(model, "model.h5")
```

Listing 15.7: Another function in Keras to save a model

How to Load a Keras Model

Your saved model can then be loaded later by calling the `load_model()` function and passing the filename. The function returns the model with the same architecture and weights. In this case, you load the model, summarize the architecture, and evaluate it on the same dataset to confirm the weights and architecture are the same.

```
# load and evaluate a saved model
from numpy import loadtxt
from tensorflow.keras.models import load_model

# load model
model = load_model('model.h5')
# summarize model.
model.summary()
# load dataset
dataset = loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# evaluate the model
score = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], score[1]*100))
```

Listing 15.8: Example of loading weights and architecture from single file

Running the example first loads the model, prints a summary of the model architecture, and then evaluates the loaded model on the same dataset. The model achieves the same accuracy score, which in this case is 77%.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 12)                108
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 104
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 9
=================================================================
Total params: 221
Trainable params: 221
Non-trainable params: 0
_________________________________________________________________

acc: 77.73%
```

Output 15.4: Sample output from saving weights and architecture to a single file

Protocol Buffer Format

While saving and loading a Keras model using HDF5 format is the recommended way, TensorFlow supports yet another format, the *protocol buffer*. It is considered faster to save and load a protocol buffer format but doing so will produce multiple files. The syntax is the same, except that you do not need to provide the `.h5` extension to the filename:

```
# save model and architecture to single file
model.save("model")

# ... later

# load model
model = load_model('model')
# print summary
model.summary()
```

Listing 15.9: Saving a Keras model in protocol buffer format

These will create a directory "`model`" with the following files:

```
model/
|-- assets/
|-- keras_metadata.pb
|-- saved_model.pb
`-- variables/
    |-- variables.data-00000-of-00001
    `-- variables.index
```

Output 15.5: Directory tree of the generated model files

This is also the format used to save a model in TensorFlow v1.x. You may encounter this when you download a pretrained model from TensorFlow Hub.

15.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Online Resources

How can I save a Keras model? Keras FAQ.
https://keras.io/getting-started/faq/%5C#how-can-i-save-a-keras-model

The Models API. Keras API Reference.
https://keras.io/api/models/

TensorFlow Hub.
https://www.tensorflow.org/hub

Protocol buffers.
https://developers.google.com/protocol-buffers/

15.5 Summary

Saving and loading models is an important capability for transplanting a deep learning model from research and development to operations. In this chapter you discovered how to serialize your Keras deep learning models. You learned:

- How to save model weights to HDF5 formatted files and load them again later.
- How to save Keras model definitions to JSON files and load them again.
- How to save Keras model definitions to YAML files and load them again.

In the next chapter, we will see how we can make Keras save the model automatically during training.

16 Keep the Best Models During Training with Checkpointing

Deep learning models can take hours, days, or even weeks to train. If the run is stopped unexpectedly, you can lose a lot of work. In this chapter, you will discover how to checkpoint your deep learning models during training in Python using the Keras library. After completing this chapter, you will know:

- ▷ The importance of checkpointing neural network models when training.
- ▷ How to checkpoint each improvement to a model during training.
- ▷ How to checkpoint the very best model observed during training.

Let's get started.

Overview

This chapter is in five parts; they are:

- ▷ Checkpointing Neural Network Models
- ▷ Checkpoint Neural Network Model Improvements
- ▷ Checkpoint the Best Neural Network Model Only
- ▷ Use EarlyStopping Together with Checkpoint
- ▷ Loading a Saved Neural Network Model

16.1 Checkpointing Neural Network Models

Application checkpointing is a fault tolerance technique for long-running processes. In this approach, a snapshot of the state of the system is taken in case of system failure. If there is a problem, not all is lost. The checkpoint may be used directly or as the starting point for a new run, picking up where it left off. When training deep learning models, the checkpoint captures the weights of the model. These weights can be used to make predictions as-is or as the basis for ongoing training.

The Keras library provides a checkpointing capability by a callback API. The `ModelCheckpoint` callback class allows you to define where to checkpoint the model weights, how to name the file, and under what circumstances to make a checkpoint of the model. The API allows you to specify which metric to monitor, such as loss or accuracy on the training or validation dataset. You can specify whether to look for an improvement in maximizing or minimizing the score. Finally, the filename you use to store the weights can include variables like the epoch number or metric. The `ModelCheckpoint` can then be passed to the training process when calling the `fit()` function on the model. Note that you may need to install the h5py library to output network weights in HDF5 format.

16.2 Checkpoint Neural Network Model Improvements

A good use of checkpointing is to output the model weights each time an improvement is observed during training. The example below creates a small neural network for the Pima Indians onset of diabetes binary classification problem (see Section 7.1). The example assumes that the `pima-indians-diabetes.csv` file is in your working directory. The example uses 33% of the data for validation.

Checkpointing is set up to save the network weights only when there is an improvement in classification accuracy on the validation dataset (`monitor='val_accuracy'` and `mode='max'`). The weights are stored in a file that includes the score in the filename, `weights-improvement-{val_accuracy:.2f}.hdf5`.

```
# Checkpoint the weights when validation accuracy improves
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
tf.random.set_seed(42)
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# checkpoint
filepath="weights-improvement-{epoch:02d}-{val_accuracy:.2f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1,
                             save_best_only=True, mode='max')
# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=150, batch_size=10,
          callbacks=[checkpoint], verbose=0)
```

Listing 16.1: Checkpoint model improvements

Running the example produces the output below, truncated for brevity. In the output you can see cases where an improvement in the model accuracy on the validation dataset resulted in a new weight file being written to disk.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
...
Epoch 00139: val_accuracy did not improve
Epoch 00140: val_accuracy improved from 0.83465 to 0.83858, saving model to
    weights-improvement-140-0.84.hdf5
Epoch 00141: val_accuracy did not improve
Epoch 00142: val_accuracy did not improve
Epoch 00143: val_accuracy did not improve
Epoch 00144: val_accuracy did not improve
Epoch 00145: val_accuracy did not improve
Epoch 00146: val_accuracy improved from 0.83858 to 0.84252, saving model to
    weights-improvement-146-0.84.hdf5
Epoch 00147: val_accuracy did not improve
Epoch 00148: val_accuracy improved from 0.84252 to 0.84252, saving model to
    weights-improvement-148-0.84.hdf5
Epoch 00149: val_accuracy did not improve
```

Output 16.1: Sample output from checkpoint model improvements

You will see a number of files in your working directory containing the network weights in HDF5 format. For example:

```
...
weights-improvement-53-0.76.hdf5
weights-improvement-71-0.76.hdf5
weights-improvement-77-0.78.hdf5
weights-improvement-99-0.78.hdf5
```

This is a very simple checkpointing strategy. It may create a lot of unnecessary checkpoint files if the validation accuracy moves up and down over training epochs. Nevertheless, it will ensure you have a snapshot of the best model discovered during your run.

16.3 Checkpoint the Best Neural Network Model Only

A simpler checkpoint strategy is to save the model weights to the same file if and only if the validation accuracy improves. This can be done easily using the same code from above and changing the output filename to be fixed (not including score or epoch information). In this case, model weights are written to the file `weights.best.hdf5` only if the classification accuracy of the model on the validation dataset improves over the best seen so far.

```
# Checkpoint the weights for best model on validation accuracy
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
import numpy as np
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# checkpoint
filepath="weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1,
                             save_best_only=True, mode='max')
callbacks_list = [checkpoint]
# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=150, batch_size=10,
          callbacks=callbacks_list, verbose=0)
```

Listing 16.2: Checkpoint best model only

Running this example provides the following output (truncated for brevity):

```
...
Epoch 00139: val_accuracy improved from 0.79134 to 0.79134, saving model to weights.best.hdf5
Epoch 00140: val_accuracy did not improve
Epoch 00141: val_accuracy did not improve
Epoch 00142: val_accuracy did not improve
Epoch 00143: val_accuracy did not improve
Epoch 00144: val_accuracy improved from 0.79134 to 0.79528, saving model to weights.best.hdf5
Epoch 00145: val_accuracy improved from 0.79528 to 0.79528, saving model to weights.best.hdf5
Epoch 00146: val_accuracy did not improve
Epoch 00147: val_accuracy did not improve
Epoch 00148: val_accuracy did not improve
Epoch 00149: val_accuracy did not improve
```

Output 16.2: Sample output from checkpoint the best model

You should see the weight file, `weights.best.hdf5`, in your local directory. This is a handy checkpoint strategy to always use during your experiments.

It will ensure that your best model is saved for the run for you to use later if you wish. It avoids needing to include any code to manually keep track and serialize the best model when training.

16.4 Use EarlyStopping Together with Checkpoint

In the examples above, an attempt was made to fit your model with 150 epochs. In reality, it is not easy to tell how many epochs you need to train your model. One way to address this problem is to overestimate the number of epochs. But this may take significant time. After all, if you are checkpointing the best model only, you may find that over the several thousand epochs run, we already achieved the best model in the first hundred epochs, and no more checkpoints are made afterward.

It is quite common to use the `ModelCheckpoint` callback together with `EarlyStopping`. It helps to stop the training once no metric improvement is seen for several epochs. The example below adds the callback `es` to make the training stop early once it does not see the validation accuracy improve for five consecutive epochs:

```python
# Checkpoint the weights for best model on validation accuracy
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
import matplotlib.pyplot as plt
import numpy as np
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# checkpoint
filepath="weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1,
                             save_best_only=True, mode='max')
es = EarlyStopping(monitor='val_accuracy', patience=5)
callbacks_list = [checkpoint, es]
# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=150, batch_size=10,
          callbacks=callbacks_list, verbose=0)
```

Listing 16.3: Checkpoint best model with early stopping

Running this example provides the following output:

```
Epoch 1: val_accuracy improved from -inf to 0.51969, saving model to weights.best.hdf5
Epoch 2: val_accuracy did not improve from 0.51969
Epoch 3: val_accuracy improved from 0.51969 to 0.54724, saving model to weights.best.hdf5
Epoch 4: val_accuracy improved from 0.54724 to 0.61417, saving model to weights.best.hdf5
Epoch 5: val_accuracy did not improve from 0.61417
Epoch 6: val_accuracy did not improve from 0.61417
Epoch 7: val_accuracy improved from 0.61417 to 0.66142, saving model to weights.best.hdf5
Epoch 8: val_accuracy did not improve from 0.66142
Epoch 9: val_accuracy did not improve from 0.66142
Epoch 10: val_accuracy improved from 0.66142 to 0.68504, saving model to weights.best.hdf5
Epoch 11: val_accuracy did not improve from 0.68504
Epoch 12: val_accuracy did not improve from 0.68504
Epoch 13: val_accuracy did not improve from 0.68504
Epoch 14: val_accuracy did not improve from 0.68504
Epoch 15: val_accuracy improved from 0.68504 to 0.69685, saving model to weights.best.hdf5
Epoch 16: val_accuracy improved from 0.69685 to 0.71260, saving model to weights.best.hdf5
Epoch 17: val_accuracy improved from 0.71260 to 0.72047, saving model to weights.best.hdf5
Epoch 18: val_accuracy did not improve from 0.72047
Epoch 19: val_accuracy did not improve from 0.72047
Epoch 20: val_accuracy did not improve from 0.72047
Epoch 21: val_accuracy did not improve from 0.72047
Epoch 22: val_accuracy did not improve from 0.72047
```

Output 16.3: Sample output from checkpoint the best model with early stopping

This training process stopped after epoch 22 as there are no better accuracy achieved for the last 5 epochs.

16.5 Loading a Saved Neural Network Model

Now that you have seen how to checkpoint your deep learning models during training, you need to review how to load and use a checkpointed model. The checkpoint only includes the model weights. It assumes you know the network structure. This, too, can be serialized to a file in JSON or YAML format. In the example below, the model structure is known, and the best weights are loaded from the previous experiment, stored in the working directory in the `weights.best.hdf5` file. The model is then used to make predictions on the entire dataset.

```python
# How to load and use weights from a checkpoint
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
import numpy as np
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# load weights
model.load_weights("weights.best.hdf5")
# Compile model (required to make predictions)
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print("Created model and loaded weights from file")
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
```

```
Y = dataset[:,8]
# estimate accuracy on whole dataset using loaded weights
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Listing 16.4: Load and evaluate a model checkpoint

Running the example produces the following output:

```
Created model and loaded weights from file
acc: 77.73%
```

Output 16.4: Sample output from loading and evaluating a model checkpoint

16.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

APIs

ModelCheckpoint. Keras API Reference.
https://keras.io/api/callbacks/model_checkpoint/

Callbacks API. Keras API Reference.
https://keras.io/api/callbacks/

16.7 Summary

In this chapter, you discovered the importance of checkpointing deep learning models for long training runs. You learned:

- ▷ How to use Keras to checkpoint each time an improvement to the model is observed.
- ▷ How to only checkpoint the very best model observed during training.
- ▷ How to load a checkpointed model from file and use it later to make predictions.

In the next chapter, you will learn how to visualize the training progress.

Understand Model Behavior During Training by Plotting History

17

You can learn a lot about neural networks and deep learning models by observing their performance over time during training. For example, if you see the training accuracy went worse with training epochs, you know you have issue with the optimization. Probably your learning rate is too fast. In this chapter, you will discover how you can review and visualize the performance of deep learning models over time during training in Python with Keras. After completing this chapter, you will know:

- ▷ How to inspect the history metrics collected during training
- ▷ How to plot accuracy metrics on training and validation datasets during training
- ▷ How to plot model loss metrics on training and validation datasets during training

Let's get started.

Overview

This chapter is in two parts; they are:

- ▷ Access Model Training History in Keras
- ▷ Visualize Model Training History in Keras

17.1 Access Model Training History in Keras

Keras provides the capability to register callbacks when training a deep learning model. One of the default callbacks registered when training all deep learning models is the History callback. It records training metrics for each epoch. This includes the loss and the accuracy (for classification problems) and the loss and accuracy for the validation dataset if one is set.

The history object is returned from calls to the `fit()` function used to train the model. Metrics are stored in a dictionary in the `history` member of the object returned. For example, you can list the metrics collected in a history object using the following snippet of code after a model is trained:

```
...
# list all data in history
print(history.history.keys())
```

Listing 17.1: Display recorded history metric names

For example, for a model trained on a classification problem with a validation dataset, this might produce the following listing:

```
['accuracy', 'loss', 'val_accuracy', 'val_loss']
```

Output 17.1: Sample output from recorded history metric names

Usually, when you called `compile()` with your Keras model, you provided the loss function as well as other metrics. The loss function is the objective to optimize. Training a neural network is to minimize the loss. The metrics are just for reporting but usually useful for the user of the network. In case of classification, "accuracy" is commonly used as metric while cross-entropy is used as the loss function. You will learn more about the loss function in Chapter 19.

You can use the data collected in the history object to create plots. The plots can provide an indication of useful things about the training of the model, such as:

- ▷ Its speed of convergence over epochs (slope)
- ▷ Whether the model may have already converged (plateau of the line)
- ▷ Whether the model may be over-learning the training data (inflection for validation line)

17.2 Visualize Model Training History in Keras

You can create plots from the collected history data. In the example below, a small network to model the Pima Indians onset of diabetes binary classification problem (see Section 7.1).

The example collects the history returned from training the model and creates two charts:

1. A plot of accuracy on the training and validation datasets over training epochs
2. A plot of loss on the training and validation datasets over training epochs

```
# Visualize training history
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
import numpy as np
# load pima indians dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
history = model.fit(X, Y, validation_split=0.33, epochs=150, batch_size=10, verbose=0)
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Listing 17.2: Evaluate a model and plot training history

The plots are provided below. The history for the validation dataset is labeled test by convention as it is indeed a test dataset for the model. From the plot of the accuracy, you can see that the model could probably be trained a little more as the trend for accuracy on both datasets is still rising for the last few epochs. You can also see that the model has not yet over-learned the training dataset, showing comparable skill on both datasets.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

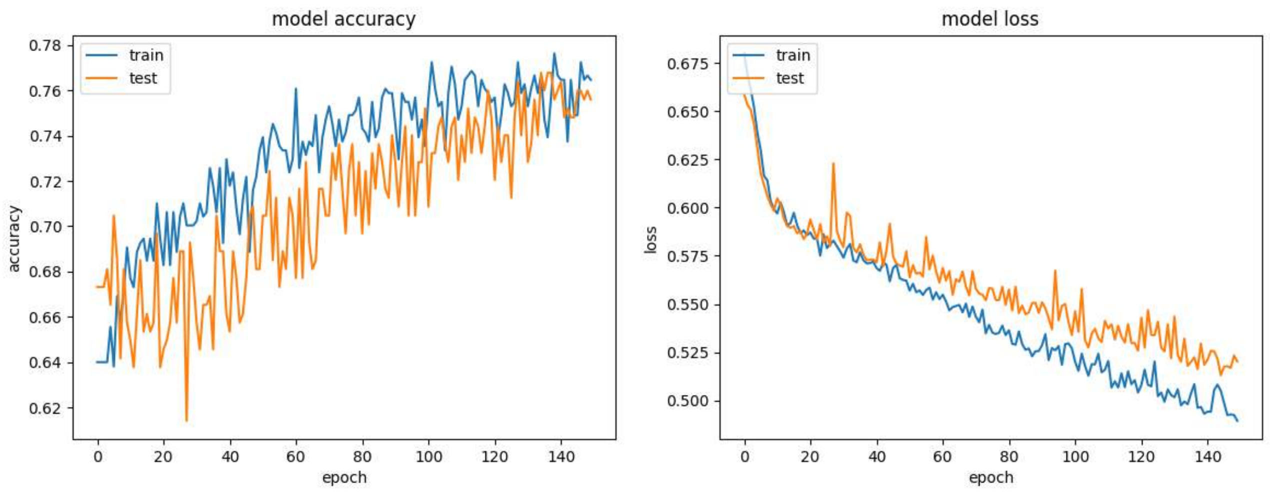


Figure 17.1: Plot of model accuracy (left) and loss (right) on training and validation datasets

From the plot of the loss, you can see that the model has comparable performance on both train and validation datasets (labeled test). If these parallel plots start to depart consistently, it might be a sign to stop training at an earlier epoch.

17.3 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

APIs

tf.keras.callbacks.History. TensorFlow API.
https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/History

17.4 Summary

In this chapter, you discovered the importance of collecting and reviewing metrics while training your deep learning models. You learned:

- ▷ How to inspect a history object returned from training to discover the metrics that were collected.
- ▷ How to extract model accuracy information for training and validation datasets and plot the data.
- ▷ How to extract and plot the model loss information calculated from training and validation datasets.

In the next chapter, you will learn about the different activation functions available in Keras.

18 Using Activation Functions in Neural Networks

Activation functions play an integral role in neural networks by introducing nonlinearity. This nonlinearity allows neural networks to develop complex representations and functions based on the inputs that would not be possible with a simple linear regression model.

Many different nonlinear activation functions have been proposed throughout the history of neural networks. In this chapter, you will explore three popular ones: sigmoid, tanh, and ReLU.

After reading this chapter, you will learn:

- ▷ Why nonlinearity is important in a neural network
- ▷ How different activation functions can contribute to the vanishing gradient problem
- ▷ Sigmoid, tanh, and ReLU activation functions
- ▷ How to use different activation functions in your TensorFlow model

Let's get started.

Overview

This chapter is divided into five parts; they are:

- ▷ Why do we need nonlinear activation functions
- ▷ Sigmoid function and vanishing gradient
- ▷ Hyperbolic tangent function
- ▷ Rectified Linear Unit (ReLU)
- ▷ Using the activation functions in practice

18.1 Why Do We Need Nonlinear Activation Functions

You might be wondering, why all this hype about nonlinear activation functions? Or why can't we just use an identity function after the weighted linear combination of activations from the previous layer? Using multiple linear layers is basically the same as using a single linear layer. This can be seen through a simple example. Let's say you have a one hidden layer neural network, each with two hidden neurons.

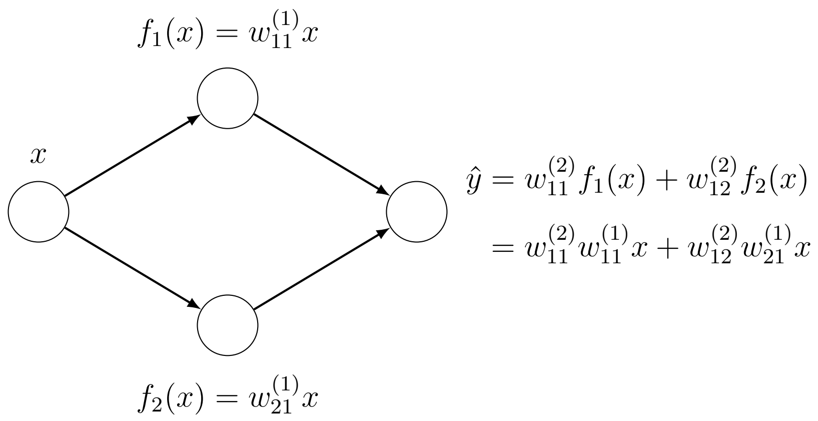

$f_1(x) = w_{11}^{(1)}x$
x
$\hat{y} = w_{11}^{(2)}f_1(x) + w_{12}^{(2)}f_2(x)$
$= w_{11}^{(2)}w_{11}^{(1)}x + w_{12}^{(2)}w_{21}^{(1)}x$
$f_2(x) = w_{21}^{(1)}x$

Figure 18.1: Single hidden layer neural network with linear layers

You can then rewrite the output layer as a linear combination of the original input variable if you used a linear hidden layer. If you had more neurons and weights, the equation would be a lot longer with more nesting and more multiplications between successive layer weights. However, the idea remains the same: You can represent the entire network as a single linear layer. To make the network represent more complex functions, you would need nonlinear activation functions. Let's start with a popular example, the sigmoid function.

18.2 Sigmoid Function and Vanishing Gradient

The sigmoid activation function is a popular choice for the nonlinear activation function for neural networks. One reason it's popular is that it has output values between 0 and 1, which mimic probability values. Hence it is used to convert the real-valued output of a linear layer to a probability, which can be used as a probability output. This also makes it an important part of logistic regression methods, which can be used directly for binary classification.

The sigmoid function is commonly represented by σ and has the form $\sigma = \dfrac{1}{1+e^{-x}}$. In TensorFlow, you can call the sigmoid function from the Keras library as follows:

```python
import tensorflow as tf
from tensorflow.keras.activations import sigmoid

input_array = tf.constant([-1, 0, 1], dtype=tf.float32)
print(sigmoid(input_array))
```

Listing 18.1: Calling a sigmoidal function

This gives the following output:

```
tf.Tensor([0.26894143 0.5        0.7310586 ], shape=(3,), dtype=float32)
```

Output 18.1: Output of a sigmoidal function

You can also plot the sigmoid function as a function of x. When looking at the activation function for the neurons in a neural network, you should also be interested in its derivative due to backpropagation and the chain rule, which would affect how the neural network learns from data.

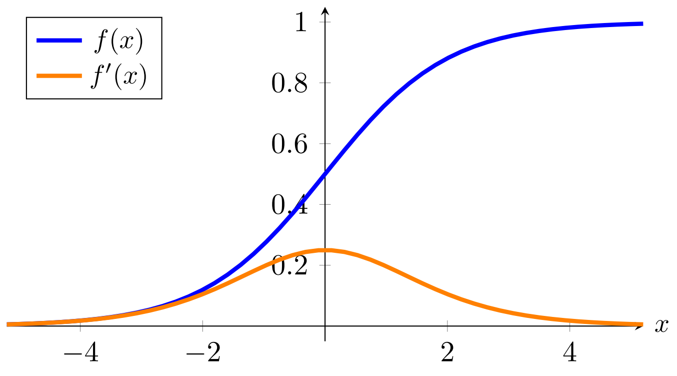


Figure 18.2: Sigmoid activation function (blue) and gradient (orange)

Here, you can observe that the gradient of the sigmoid function is always between 0 and 0.25. And as the x tends to positive or negative infinity, the gradient tends to zero. This could contribute to the vanishing gradient problem, meaning when the inputs are at some large magnitude of x (e.g., due to the output from earlier layers), the gradient is too small to initiate the correction.

Vanishing gradient is a problem because the chain rule is used in backpropagation in deep neural networks. Recall that in neural networks, the gradient (of the loss function) at each layer is the gradient at its subsequent layer multiplied by the gradient of its activation function. As there are many layers in the network, if the gradient of the activation functions is less than 1, the gradient at some layer far away from the output will be close to zero. And any layer with a gradient close to zero will stop the gradient propagation further back to the earlier layers.

Since the sigmoid function is always less than 1, a network with more layers would exacerbate the vanishing gradient problem. Furthermore, there is a saturation region where the gradient of the sigmoid tends to 0, which is where the magnitude of x is large. So, if the output of the weighted sum of activations from previous layers is large, then you would have a very small gradient propagating through this neuron, as the derivative of the activation a with respect to the input to the activation function would be small (in the saturation region).

Granted, there is also the derivative of the linear term with respect to the previous layer's activations which might be greater than 1 for the layer since the weight might be large, and it's a sum of derivatives from the different neurons. However, it might still raise concern at the start of training as weights are usually initialized to be small.

18.3 Hyperbolic Tangent Function

Another activation function to consider is the tanh activation function, also known as the hyperbolic tangent function. It has a larger range of output values compared to the sigmoid function and a larger maximum gradient. The tanh function is a hyperbolic analog to the normal tangent function for circles that most people are familiar with.

Plotting out the tanh function and its gradient:

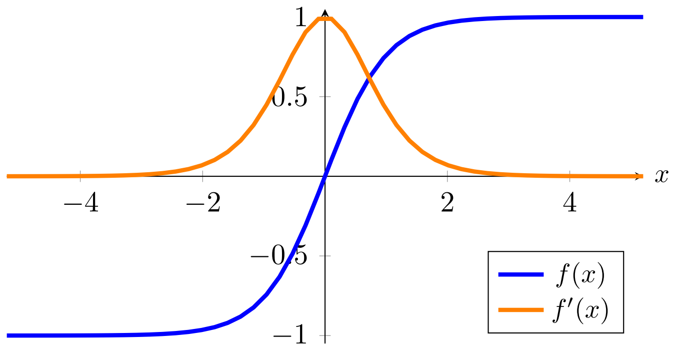


Figure 18.3: Tanh activation function (blue) and gradient (orange)

Notice that the gradient now has a maximum value of 1, compared to the sigmoid function, where the largest gradient value is 0.25. This makes a network with tanh activation less susceptible to the vanishing gradient problem. However, the tanh function also has a saturation region, where the value of the gradient tends toward as the magnitude of the input x gets larger.

In TensorFlow, you can implement the tanh activation on a tensor using the `tanh` function in Keras' activations module:

```python
import tensorflow as tf
from tensorflow.keras.activations import tanh

input_array = tf.constant([-1, 0, 1], dtype=tf.float32)
print(tanh(input_array))
```

Listing 18.2: Calling a hyperbolic tangent function

This gives the output:

```
tf.Tensor([-0.7615942  0.         0.7615942], shape=(3,), dtype=float32)
```

Output 18.2: Output of the tanh function

18.4 Rectified Linear Unit (ReLU)

The last activation function to cover in detail is the Rectified Linear Unit, also popularly known as ReLU. It has become popular recently due to its relatively simple computation. This helps to speed up neural networks and seems to get empirically good performance, which makes it a good starting choice for the activation function.

The ReLU function is a simple $\max(0, x)$ function, which can also be thought of as a piecewise function with all inputs less than 0 mapping to 0 and all inputs greater than or equal to 0 mapping back to themselves (i.e., identity function). Graphically,

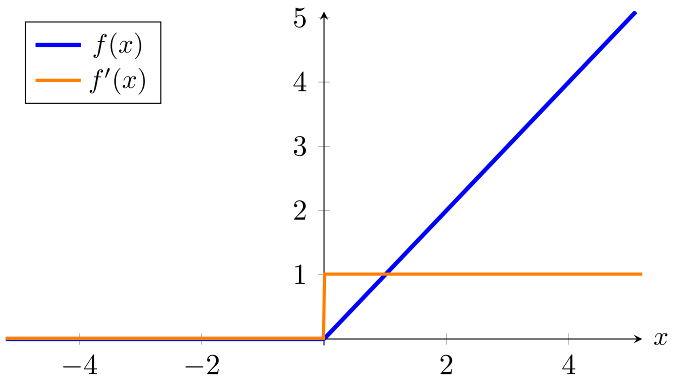


Figure 18.4: ReLU activation function (blue) and gradient (orange)

Notice that the gradient of ReLU is 1 whenever the input is positive, which helps address the vanishing gradient problem. However, whenever the input is negative, the gradient is 0. This can cause another problem, the dead neuron/dying ReLU problem, which is an issue if a neuron is *persistently inactivated*. In this case, the neuron can never learn, and its weights are never updated due to the chain rule as it has a 0 gradient as one of its terms. If this happens for all data in your dataset, then it can be very difficult for this neuron to learn from your dataset unless the activations in the previous layer change such that the neuron is no longer "dead".

To use the ReLU activation in TensorFlow:

```
import tensorflow as tf
from tensorflow.keras.activations import relu

input_array = tf.constant([-1, 0, 1], dtype=tf.float32)
print(relu(input_array))
```

Listing 18.3: Calling a rectified linear unit function

This gives the following output:

```
tf.Tensor([0. 0. 1.], shape=(3,), dtype=float32)
```

Output 18.3: Output of the ReLU function

The three activation functions reviewed above show that they are all monotonically increasing functions. This is required; otherwise, you cannot apply the gradient descent algorithm.

Now that you've explored some common activation functions and how to use them in TensorFlow. Let's take a look at how you can use these in practice in an actual model.

18.5 Using Activation Functions in Practice

Before exploring the use of activation functions in practice, let's look at another common way to use activation functions when combining them with another Keras layer. Let's say you want to add a ReLU activation on top of a `Dense` layer. One way you can do this following the above methods shown is to do:

```
x = Dense(units=10)(input_layer)
x = relu(x)
```

Listing 18.4: Connecting an activation function to a Keras layer

However, for many Keras layers, you can also use a more compact representation to add the activation on top of the layer:

```
x = Dense(units=10, activation="relu")(input_layer)
```

Listing 18.5: Setting activation function to a Keras layer

Using this more compact representation, let's build our LeNet5 model using Keras:

```
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.layers import Dense, Input, Flatten, \
                                    Conv2D, BatchNormalization, MaxPool2D
from tensorflow.keras.models import Model

(trainX, trainY), (testX, testY) = keras.datasets.cifar10.load_data()

input_layer = Input(shape=(32,32,3,))
x = Conv2D(6, (5,5), padding="same", activation="relu")(input_layer)
x = MaxPool2D(pool_size=(2,2))(x)
x = Conv2D(16, (5,5), padding="same", activation="relu")(x)
x = MaxPool2D(pool_size=(2, 2))(x)
x = Conv2D(120, (5,5), padding="same", activation="relu")(x)
x = Flatten()(x)
x = Dense(units=84, activation="relu")(x)
x = Dense(units=10, activation="softmax")(x)

model = Model(inputs=input_layer, outputs=x)
model.summary()

model.compile(optimizer="adam", loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics="acc")

history = model.fit(x=trainX, y=trainY, batch_size=256, epochs=10,
                    validation_data=(testX, testY))
```

Listing 18.6: Building the LeNet5 network and train with CIFAR-10 dataset

And running this code gives the following output:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 6)         456

 max_pooling2d (MaxPooling2D)  (None, 16, 16, 6)       0

 conv2d_1 (Conv2D)             (None, 16, 16, 16)          2416

 max_pooling2d_1 (MaxPooling2D)  (None, 8, 8, 16)          0

 conv2d_2 (Conv2D)             (None, 8, 8, 120)           48120

 flatten (Flatten)             (None, 7680)                0

 dense (Dense)                 (None, 84)                  645204

 dense_1 (Dense)               (None, 10)                  850

=================================================================
Total params: 697,046
Trainable params: 697,046
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
196/196 [==============================] - 14s 11ms/step - loss: 2.9758 acc: 0.3390 - val_loss: 1.55
Epoch 2/10
196/196 [==============================] - 2s 8ms/step - loss: 1.4319 - acc: 0.4927 - val_loss: 1.38
Epoch 3/10
196/196 [==============================] - 2s 8ms/step - loss: 1.2505 - acc: 0.5583 - val_loss: 1.35
Epoch 4/10
196/196 [==============================] - 2s 8ms/step - loss: 1.1127 - acc: 0.6094 - val_loss: 1.28
Epoch 5/10
196/196 [==============================] - 2s 8ms/step - loss: 0.9763 - acc: 0.6594 - val_loss: 1.32
Epoch 6/10
196/196 [==============================] - 2s 8ms/step - loss: 0.8510 - acc: 0.7017 - val_loss: 1.39
Epoch 7/10
196/196 [==============================] - 2s 8ms/step - loss: 0.7361 - acc: 0.7426 - val_loss: 1.41
Epoch 8/10
196/196 [==============================] - 2s 8ms/step - loss: 0.6060 - acc: 0.7894 - val_loss: 1.53
Epoch 9/10
196/196 [==============================] - 2s 8ms/step - loss: 0.5020 - acc: 0.8265 - val_loss: 1.78
Epoch 10/10
196/196 [==============================] - 2s 8ms/step - loss: 0.4013 - acc: 0.8605 - val_loss: 1.83
```

Output 18.4: Output of training a LeNet5 network

And that's how you can use different activation functions in your TensorFlow models!

18.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

APIs

Layer activation functions. Keras API Reference.
    https://keras.io/api/layers/activations/

tf.keras.layers.ReLU. TensorFlow API.
    https://www.tensorflow.org/api_docs/python/tf/keras/layers/ReLU

tf.keras.layers.LeakyReLU. TensorFlow API.
https://www.tensorflow.org/api_docs/python/tf/keras/layers/LeakyReLU

Papers

Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *Proceedings of IEEE International Conference on Computer Vision (ICCV)*. 2015.
https://arxiv.org/abs/1502.01852

Ian J. Goodfellow et al. "Maxout Networks". In: *Proceedings of the 30th International Conference on Machine Learning*. Atlanta, GA, USA, 2013.
https://arxiv.org/abs/1302.4389

18.7 Summary

In this chapter, you have seen why activation functions are important to allow for the complex neural networks that are common in deep learning today. You have also seen some popular activation functions, their derivatives, and how to integrate them into your TensorFlow models. Specifically, you learned:

▷ Why nonlinearity is important in a neural network

▷ How different activation functions can contribute to the vanishing gradient problem

▷ Sigmoid, tanh, and ReLU activation functions

▷ How to use different activation functions in your TensorFlow model

In the next chapter, you will learn about the various loss functions.

Loss Functions in TensorFlow

19

The loss metric is very important for neural networks. As all machine learning models are one optimization problem or another, the loss is the objective function to minimize. In neural networks, the optimization is done with gradient descent and backpropagation. But what are loss functions, and how are they affecting your neural networks?

In this chapter, you will learn what loss functions are and delve into some commonly used loss functions and how you can apply them to your neural networks. After reading this chapter, you will learn:

- ▷ What are loss functions, and how they are different from metrics
- ▷ Common loss functions for regression and classification problems
- ▷ How to use loss functions in your TensorFlow model

Let's get started!

Overview

This chapter is divided into five sections; they are:

- ▷ What Are Loss Functions?
- ▷ Mean Absolute Error
- ▷ Mean Squared Error
- ▷ Categorical Cross-Entropy
- ▷ Loss Functions in Practice

19.1 What Are Loss Functions?

In neural networks, loss functions help optimize the performance of the model. They are usually used to measure some penalty that the model incurs on its predictions, such as the deviation of the prediction away from the ground truth label. Loss functions are usually differentiable across their domain (but it is allowed that the gradient is undefined only for very specific points, such as $x = 0$, which is basically ignored in practice). In the training loop, they are differentiated with respect to parameters, and these gradients are used for your backpropagation and gradient descent steps to optimize your model on the training set.

Loss functions are also slightly different from metrics. While loss functions can tell you the performance of our model, they might not be of direct interest or easily explainable by humans. This is where metrics come in. Metrics such as accuracy are much more useful for humans to understand the performance of a neural network even though they might not be good choices for loss functions since they might not be differentiable.

In the following, let's explore some common loss functions: the mean absolute error, mean squared error, and categorical cross-entropy.

19.2 Mean Absolute Error

The mean absolute error (MAE) measures the absolute difference between predicted values and the ground truth labels and takes the mean of the difference across all training examples. Mathematically, it is equal to $\frac{1}{m}\sum_{i=1}^{m}|\hat{y}_i - y_i|$ where m is the number of training examples y_i and $\hat{y}_i$ are the ground truth and predicted values, respectively, averaged over all training examples. The MAE is never negative and would be zero only if the prediction matched the ground truth perfectly. It is an intuitive loss function and might also be used as one of your metrics, specifically for regression problems, since you want to minimize the error in your predictions.

Let's look at what the mean absolute error loss function looks like graphically:

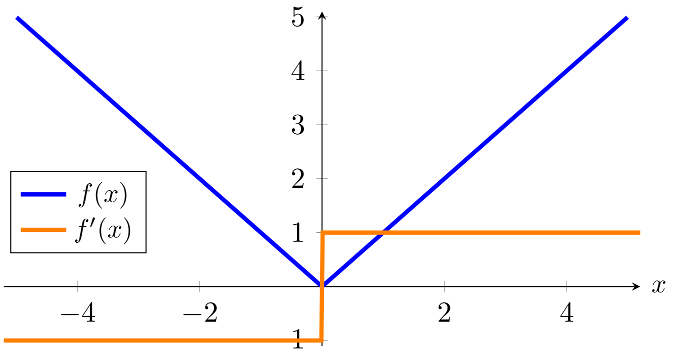


Figure 19.1: Mean absolute error loss function (blue) and gradient (orange)

Similar to activation functions, you might also be interested in what the gradient of the loss function looks like since you are using the gradient later to do backpropagation to train your model's parameters.

You might notice a discontinuity in the gradient function for the mean absolute loss function. Many tend to ignore it since it occurs only at $x = 0$, which, in practice, rarely happens since it is the probability of a single point in a continuous distribution.

Let's take a look at how to implement this loss function in TensorFlow using the Keras losses module:

```
import tensorflow as tf
from tensorflow.keras.losses import MeanAbsoluteError

y_true = [1., 0.]
y_pred = [2., 3.]

mae_loss = MeanAbsoluteError()

print(mae_loss(y_true, y_pred).numpy())
```

Listing 19.1: Compute the mean absolute error

This gives you `2.0` as the output as expected, since $\frac{1}{2}(|2-1|+|3-0|) = \frac{1}{2}(4) = 4$. Next, let's explore another loss function for regression models with slightly different properties, the mean squared error.

19.3 Mean Squared Error

Another popular loss function for regression models is the mean squared error (MSE), which is equal to $\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$. It is similar to the mean absolute error as it also measures the deviation of the predicted value from the ground truth value. However, the mean squared error squares this difference (always non-negative since squares of real numbers are always non-negative), which gives it slightly different properties.

One property is that the mean squared error favors a large number of small errors over a small number of large errors, which leads to models with fewer outliers or at least outliers that are less severe than models trained with a mean absolute error. This is because a large error would have a significantly larger impact on the error and, consequently, the gradient of the error when compared to a small error. Graphically,

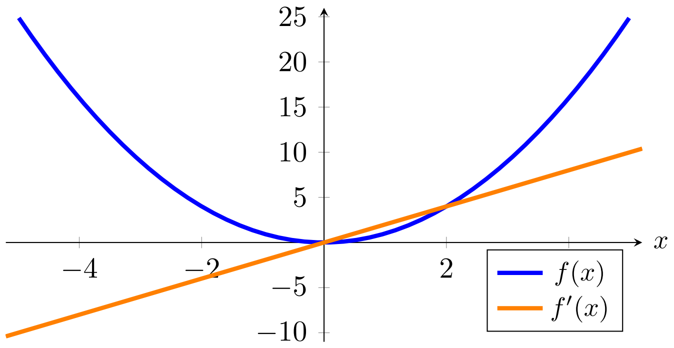


Figure 19.2: Mean square error loss function (blue) and gradient (orange)

Notice that larger errors would lead to a larger magnitude for the gradient and a larger loss. Hence, for example, two training examples that deviate from their ground truths by 1 unit would lead to a loss of 2, while a single training example that deviates from its ground truth by 2 units would lead to a loss of 4, hence having a larger impact.

Let's look at how to implement the mean squared loss in TensorFlow.

```
import tensorflow as tf
from tensorflow.keras.losses import MeanSquaredError

y_true = [1., 0.]
y_pred = [2., 3.]

mse_loss = MeanSquaredError()

print(mse_loss(y_true, y_pred).numpy())
```

Listing 19.2: Compute the mean squared error

This gives the output `5.0` as expected since $\frac{1}{2}[(2-1)^2+(3-0)^2] = \frac{1}{2}(10) = 5$. Notice that the second example with a predicted value of 3 and actual value of 0 contributes 90% of the error under the mean squared error vs. 75% under the mean absolute error.

Sometimes, you may see people use root mean squared error (RMSE) as a metric. This will take the square root of MSE. From the perspective of a loss function, MSE and RMSE are equivalent.

Both MAE and MSE measure values in a continuous range. Hence they are for regression problems. For classification problems, you can use categorical cross-entropy.

19.4 Categorical Cross-Entropy

The previous two loss functions are for regression models, where the output could be any real number. However, for classification problems, there is a small, discrete set of numbers that the output could take. Furthermore, the number used to label-encode the classes is arbitrary and with no semantic meaning (e.g., using the labels 0 for cat, 1 for dog, and 2 for horse does not represent that a dog is half cat and half horse). Therefore, it should not have an impact on the performance of the model.

In a classification problem, the model's output is a vector of probability for each category. In Keras models, this vector is usually expected to be "logits," i.e., real numbers to be transformed to probability using the softmax function, or the output of a softmax activation function.

The cross-entropy between two probability distributions is a measure of the difference between the two probability distributions. Precisely, it is $-\sum_i P(X=x_i)\log Q(X=x_i)$ for probability P and Q. In machine learning, we usually have the probability P provided by the training data and Q predicted by the model, which P is 1 for the correct class and 0 for every other class. The predicted probability Q, however, is usually valued between 0 and 1. Hence when used for classification problems in machine learning, this formula can be simplified into:

$$\text{categorical cross-entropy} = -\log p_{gt}$$

where p_{gt} is the model-predicted probability of the ground truth class for that particular sample.

Cross-entropy metrics have a negative sign because $\log(x)$ tends to negative infinity as x tends to zero. We want a higher loss when the probability approaches 0 and a lower loss when the probability approaches 1. Graphically,

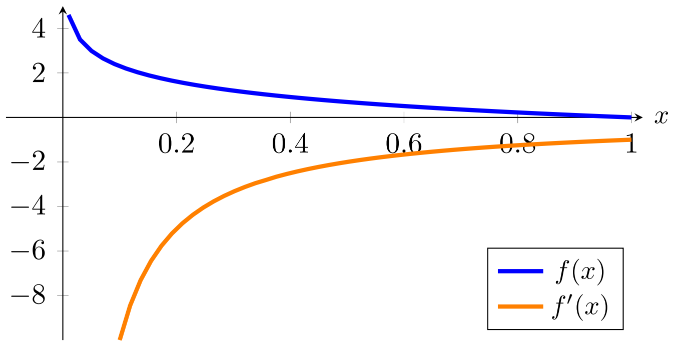


Figure 19.3: Categorical cross-entropy loss function (blue) and gradient (orange)

Notice that the loss is exactly 0 if the probability of the ground truth class is 1 as desired. Also, as the probability of the ground truth class tends to 0, the loss tends to positive infinity as well, hence substantially penalizing bad predictions. You might recognize this loss function for logistic regression, which is similar except the logistic regression loss is specific to the case of binary classes.

Looking at the gradient, you can see that the gradient is generally negative, which is also expected since, to decrease this loss, you would want the probability on the ground truth class to be as high as possible. Recall that gradient descent goes in the opposite direction of the gradient.

There are two different ways to implement categorical cross-entropy in TensorFlow. The first method takes in one-hot vectors as input,

```python
import tensorflow as tf
from tensorflow.keras.losses import CategoricalCrossentropy

# using one-hot vector representation
y_true = [[0, 1, 0], [1, 0, 0]]
y_pred = [[0.15, 0.75, 0.1], [0.75, 0.15, 0.1]]

cross_entropy_loss = CategoricalCrossentropy()

print(cross_entropy_loss(y_true, y_pred).numpy())
```

Listing 19.3: Compute the categorical cross-entropy

This gives the output as `0.2876821` which is equal to $-\log(0.75)$ as expected. The other way of implementing the categorical cross-entropy loss in TensorFlow is using a label-encoded representation for the class, where the class is represented by a single non-negative integer indicating the ground truth class instead.

```python
import tensorflow as tf
from tensorflow.keras.losses import SparseCategoricalCrossentropy

y_true = [1, 0]
y_pred = [[0.15, 0.75, 0.1], [0.75, 0.15, 0.1]]

cross_entropy_loss = SparseCategoricalCrossentropy()
```

```
print(cross_entropy_loss(y_true, y_pred).numpy())
```

Listing 19.4: Compute the sparse cross-entropy

This likewise gives the output `0.2876821`.

Now that you've explored loss functions for both regression and classification models, let's take a look at how you can use loss functions in your machine learning models.

19.5 Loss Functions in Practice

Let's explore how to use loss functions in practice. You'll explore this through a simple dense model on the MNIST digit classification dataset.

First, download the data from the Keras datasets module:

```
import tensorflow as tf

(trainX, trainY), (testX, testY) = tf.keras.datasets.mnist.load_data()
```

Listing 19.5: Getting the MNIST dataset

Then, build your model:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Input, Flatten

model = Sequential([
          Input(shape=(28,28,1,)),
          Flatten(),
          Dense(units=84, activation="relu"),
          Dense(units=10, activation="softmax"),
      ])

print (model.summary())
```

Listing 19.6: Building a model for MNIST dataset

And look at the model architecture outputted from the above code:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dense_2 (Dense)             (None, 84)                65940

 dense_3 (Dense)             (None, 10)                850

=================================================================
Total params: 66,790
Trainable params: 66,790
```

```
Non-trainable params: 0
_________________________________________________________________
```

Output 19.1: The model architecture

You can then compile your model, which is also where you introduce the loss function. Since this is a classification problem, use the cross-entropy loss. In particular, since the MNIST dataset in Keras datasets is represented as a label instead of a one-hot vector, use the `SparseCategoricalCrossEntropy` loss.

```
model.compile(optimizer="adam",
              loss=tf.keras.losses.SparseCategoricalCrossentropy(), metrics="acc")
```

Listing 19.7: Setting loss function to a model

And finally, you train your model, with the complete code below:

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Input, Flatten

(trainX, trainY), (testX, testY) = tf.keras.datasets.mnist.load_data()

model = Sequential([
          Input(shape=(28,28,1,)),
          Flatten(),
          Dense(units=84, activation="relu"),
          Dense(units=10, activation="softmax"),
      ])

model.summary()

model.compile(optimizer="adam",
              loss=tf.keras.losses.SparseCategoricalCrossentropy(), metrics="acc")

history = model.fit(x=trainX, y=trainY, batch_size=256, epochs=10,
                    validation_data=(testX, testY))
```

Listing 19.8: Build a train a model using sparse cross-entropy loss function

And your model successfully trains with the following output:

```
Epoch 1/10
235/235 [==============================] - 2s 6ms/step - loss: 7.8607 - acc: 0.8184 - val_loss: 1.74
Epoch 2/10
235/235 [==============================] - 1s 6ms/step - loss: 1.1011 - acc: 0.8854 - val_loss: 0.90
Epoch 3/10
235/235 [==============================] - 1s 6ms/step - loss: 0.5729 - acc: 0.8998 - val_loss: 0.66
Epoch 4/10
235/235 [==============================] - 1s 5ms/step - loss: 0.3911 - acc: 0.9203 - val_loss: 0.54
Epoch 5/10
235/235 [==============================] - 1s 6ms/step - loss: 0.3016 - acc: 0.9306 - val_loss: 0.50
Epoch 6/10
235/235 [==============================] - 1s 6ms/step - loss: 0.2443 - acc: 0.9405 - val_loss: 0.45
```

```
Epoch 7/10
235/235 [==============================] - 1s 5ms/step - loss: 0.2076 - acc: 0.9469 - val_loss: 0.41
Epoch 8/10
235/235 [==============================] - 1s 5ms/step - loss: 0.1852 - acc: 0.9514 - val_loss: 0.43
Epoch 9/10
235/235 [==============================] - 1s 6ms/step - loss: 0.1576 - acc: 0.9577 - val_loss: 0.42
Epoch 10/10
235/235 [==============================] - 1s 5ms/step - loss: 0.1455 - acc: 0.9597 - val_loss: 0.41
```

Output 19.2: Output of a model trained using the sparse cross-entropy loss function

And that's one example of how to use a loss function in a TensorFlow model.

19.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

APIs

Losses. Keras API Reference.
https://keras.io/api/losses/

tf.keras.losses.MeanAbsoluteError. TensorFlow API.
https://www.tensorflow.org/api_docs/python/tf/keras/losses/MeanAbsoluteError

tf.keras.losses.MeanSquaredError. TensorFlow API.
https://www.tensorflow.org/api_docs/python/tf/keras/losses/MeanSquaredError

tf.keras.losses.CategoricalCrossentropy. TensorFlow API.
https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy

tf.keras.losses.SparseCategoricalCrossentropy. TensorFlow API.
https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

19.7 Summary

In this chapter, you have seen loss functions and the role that they play in a neural network. You have also seen some popular loss functions used in regression and classification models, as well as how to use the cross-entropy loss function in a TensorFlow model.

Specifically, you learned:

▷ What are loss functions, and how they are different from metrics

▷ Common loss functions for regression and classification problems

▷ How to use loss functions in your TensorFlow model

In the next chapter, we will learn about regularization.

20 Reduce Overfitting with Dropout Regularization

Dropout is a simple and powerful regularization technique for neural networks and deep learning models. The goal is to prevent overfitting, which is when the network know too much about the training data that became ineffective to the validation data that it never saw. In this chapter, you will discover the dropout regularization technique and how to apply it to your models in Python with Keras. After reading this chapter, you will know:

▷ How the dropout regularization technique works

▷ How to use dropout on your input layers

▷ How to use dropout on your hidden layers

▷ How to tune the dropout level on your problem

Let's get started.

Overview

This chapter is divided into six parts; they are:

▷ Dropout Regularization For Neural Networks

▷ Dropout Regularization in Keras

▷ Using Dropout on the Visible Layer

▷ Using Dropout on Hidden Layers

▷ Dropout in Evaluation Mode

▷ Tips For Using Dropout

20.1 Dropout Regularization For Neural Networks

Dropout is a regularization technique for neural network models proposed by Srivastava et al. in their 2014 paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". Dropout is a technique where randomly selected neurons are ignored during training. They are *dropped out* randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass, and any weight updates are not applied to the neuron on the backward pass.

As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features, providing some specialization. Neighboring neurons come to rely on this specialization, which, if taken too far, can result in a fragile model too specialized for the training data, i.e., overfitting. This reliance context for a neuron during training is referred to as *complex co-adaptations*. You can imagine that if neurons are randomly dropped out of the network during training, other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.

The effect is that the network becomes less sensitive to the specific weights of neurons. This, in turn, results in a network capable of better generalization and less likely to overfit the training data.

20.2 Dropout Regularization in Keras

Dropout is easily implemented by randomly selecting nodes to be dropped out with a given probability (e.g., 20%) in each weight update cycle. This is how Dropout is implemented in Keras. Dropout is only used during the training of a model and is not used when evaluating the skill of the model. Next, let's explore a few different ways of using Dropout in Keras.

The examples will use the Sonar binary classification dataset (see Section 10.1). You will evaluate the developed models using scikit-learn with 10-fold cross-validation in order to tease out differences in the results better. There are 60 input values and a single output value. The input values are standardized before being used in the network. The baseline neural network model has two hidden layers, the first with 60 units and the second with 30. Stochastic gradient descent is used to train the model with a relatively low learning rate and momentum.

The full baseline model is listed below:

```python
# Baseline Model on the Sonar Dataset
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
```

```python
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# baseline
def create_baseline():
    # create model
    model = Sequential()
    model.add(Dense(60, input_shape=(60,), activation='relu'))
    model.add(Dense(30,  activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    sgd = SGD(learning_rate=0.01, momentum=0.8)
    model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
    return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_baseline,
                                          epochs=300, batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 20.1: Baseline neural network for the sonar dataset

Running the example for the baseline model generates an estimated classification accuracy of 86%.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Baseline: 86.04% (4.58%)
```

Output 20.1: Sample output from baseline neural network for the sonar dataset

20.3 Using Dropout on the Visible Layer

Dropout can be applied to input neurons called the visible layer. In the example below, a new `Dropout` layer between the input (or visible layer) and the first hidden layer was added. The dropout rate is set to 20%, meaning one in five inputs will be randomly excluded from each update cycle. In the example of the sonar dataset, using dropout means the prediction is based on only 80% of the input features (randomly chosen).

Additionally, as recommended in the original paper on dropout, a constraint is imposed on the weights for each hidden layer, ensuring that the maximum norm of the weights does not exceed a value of 3. This is done by setting the `kernel_constraint` argument on the `Dense` class when constructing the layers. The learning rate was lifted by one order of magnitude, and the momentum was increased to 0.9. These increases in the learning rate were also recommended in the original dropout paper. Continuing from the baseline example above, the code below exercises the same network with input dropout:

```
# Example of Dropout on the Sonar Dataset: Visible Layer
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.optimizers import SGD
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# dropout in the input layer with weight constraint
def create_model():
    # create model
    model = Sequential()
    model.add(Dropout(0.2, input_shape=(60,)))
    model.add(Dense(60, activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dense(30, activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    sgd = SGD(learning_rate=0.1, momentum=0.9)
    model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
    return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_model,
                                          epochs=300, batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Visible: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 20.2: Example of using dropout on the visible layer

Running the example provides a slight drop in classification accuracy, at least on a single test run.

```
Visible: 83.52% (7.68%)
```

Output 20.2: Sample output from example of using dropout on the visible layer

20.4 Using Dropout on Hidden Layers

Dropout can be applied to hidden neurons in the body of your network model. In the example below, dropout is applied between the two hidden layers and between the last hidden layer and the output layer. Again a dropout rate of 20% is used as is a weight constraint on those layers. In the example of the sonar dataset, this means the prediction is based on only 80% of the features learned by the hidden layer (randomly chosen).

```python
# Example of Dropout on the Sonar Dataset: Hidden Layer
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.optimizers import SGD
from scikeras.wrappers import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# dropout in hidden layers with weight constraint
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(60, input_shape=(60,),
                    activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(30, activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    sgd = SGD(learning_rate=0.1, momentum=0.9)
    model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
    return model

estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(model=create_model,
                                          epochs=300, batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Hidden: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 20.3: Example of using dropout on the hidden layer

You can see that for this problem and the chosen network configuration, using dropout in the hidden layers did not lift performance. In fact, performance was worse than the baseline. It is possible that additional training epochs are required or that further tuning is required to the learning rate.

```
Hidden: 83.59% (7.31%)
```

Output 20.3: Sample output from example of using dropout on the hidden layer

20.5 Dropout in Evaluation Mode

Dropout will randomly reset some of the input to zero. If you wonder what happens after you have finished training, the answer is nothing! In Keras, a layer can tell if the model is running in training mode or not. The `Dropout` layer will randomly reset some input only when the model runs for training. Otherwise, the `Dropout` layer works as a scaler to multiply all input by a factor such that the next layer will see input similar in scale. Precisely, if the dropout rate is r, the input will be scaled by a factor of $1 - r$.

20.6 Tips For Using Dropout

The original paper on Dropout provides experimental results on a suite of standard machine learning problems. As a result, they provide a number of useful heuristics to consider when using dropout in practice:

- Generally, use a small dropout value of 20%–50% of neurons, with 20% providing a good starting point. A probability too low has minimal effect, and a value too high results in under-learning by the network.
- Use a larger network. You are likely to get better performance when dropout is used on a larger network, giving the model more of an opportunity to learn independent representations.
- Use dropout on input (visible) as well as hidden layers. Application of dropout at each layer of the network has shown good results.

- ▷ Use a large learning rate with decay and a large momentum. Increase your learning rate by a factor of 10 to 100 and use a high momentum value of 0.9 or 0.99.
- ▷ Constrain the size of network weights. A large learning rate can result in very large network weights. Imposing a constraint on the size of network weights, such as max-norm regularization, with a size of 4 or 5 has been shown to improve results.

20.7 More Resources on Dropout

This section provides more resources on the topic if you are looking to go deeper.

Articles

Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. *Journal of Machine Learning Research*, 15(56), 2014, pp. 1929–1958.
https://jmlr.org/papers/v15/srivastava14a.html

Geoffrey E. Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”, 2012.
https://arxiv.org/abs/1207.0580

How does the dropout method work in deep learning? Quora.
https://www.quora.com/How-does-the-dropout-method-work-in-deep-learning

Keras Training and Evaluation with Built-in Methods. TensorFlow documentation.
https://www.tensorflow.org/guide/keras/train_and_evaluate

20.8 Summary

In this chapter, you discovered the dropout regularization technique for deep learning models. You learned:

- ▷ What dropout is and how it works
- ▷ How you can use dropout on your own deep learning models
- ▷ Tips for getting the best results from dropout on your own models

In the next chapter, you will learn how to change the learning rate during training.

Lift Performance with Learning Rate Schedules

21

Training a neural network or large deep learning model is a difficult optimization task. The classical algorithm to train neural networks is called stochastic gradient descent. It has been well established that you can achieve increased performance and faster training on some problems by using a learning rate that changes during training. In this chapter, you will discover how you can use different learning rate schedules for your neural network models in Python using the Keras deep learning library. After completing this chapter, you will know:

▷ The benefit of learning rate schedules on lifting model performance during training

▷ How to configure and evaluate a time-based learning rate schedule

▷ How to configure and evaluate a drop-based learning rate schedule

Let's get started.

Overview

This chapter is divided into four parts; they are:

▷ Learning Rate Schedule For Training Models

▷ Time-Based Learning Rate Schedule

▷ Drop-Based Learning Rate Schedule

▷ Tips for Using Learning Rate Schedules

21.1 Learning Rate Schedule for Training Models

Adapting the learning rate for your stochastic gradient descent optimization procedure can increase performance and reduce training time. Sometimes, this is called learning rate annealing or adaptive learning rates. Here, this approach is called a learning rate schedule, where the default schedule usesr a constant learning rate to update network weights for each training epoch.

The simplest and perhaps most used adaptation of the learning rates during training are techniques that reduce the learning rate over time. These have the benefit of making large changes at the beginning of the training procedure when larger learning rate values are used and decreasing the learning rate so that a smaller rate and, therefore, smaller training updates are made to weights later in the training procedure. This has the effect of quickly learning good weights early and fine-tuning them later. Two popular and easy-to-use learning rate schedules are as follows:

▷ Decrease the learning rate gradually based on the epoch

▷ Decrease the learning rate using punctuated large drops at specific epochs

Next, let's look at how you can use each of these learning rate schedules in turn with Keras.

21.2 Time-Based Learning Rate Schedule

Keras has a built-in time-based learning rate schedule. The stochastic gradient descent optimization algorithm implementation in the `SGD` class has an argument called decay. This argument is used in the time-based learning rate decay schedule equation as follows:

$$\text{LearningRate} := \text{LearningRate} \times \frac{1}{1 + \text{decay} \times \text{epoch}}$$

When the decay argument is zero (the default), this does not affect the learning rate. For example, when the learning rate was 0.1,

$$\text{LearningRate} := 0.1 \times \frac{1}{1 + 0 \times \text{epoch}} = 0.1$$

When the decay argument is specified, it will decrease the learning rate from the previous epoch by the given fixed amount. For example, if you use the initial learning rate value of 0.1 and the decay of 0.001, the first five epochs will adapt the learning rate as follows:

```
Epoch    Learning Rate
1        0.1
2        0.0999000999
3        0.0997006985
4        0.09940249103
5        0.09900646517
```

Output 21.1: Learning rate with decay as calculated

Extending this out to 100 epochs will produce the following graph of learning rate (y-axis) versus epoch (x-axis):

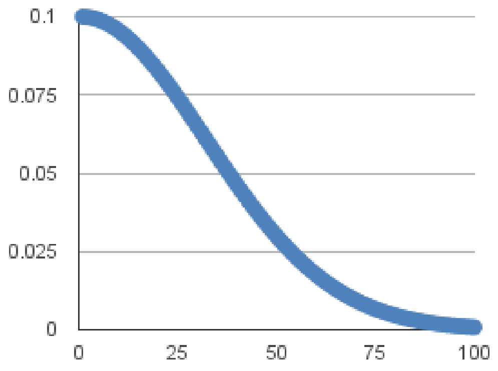


Figure 21.1: Time-based learning rate schedule

You can create a nice default schedule by setting the decay value as follows:

$$\text{Decay} = \frac{\text{LearningRate}}{\text{Epochs}} = \frac{0.1}{100} = 0.001$$

The example below demonstrates using the time-based learning rate adaptation schedule in Keras.

It is demonstrated in the Ionosphere binary classification problem. The ionosphere dataset is good for practicing with neural networks because all the input values are small numerical values of the same scale. The dataset describes radar returns where the target was free electrons in the ionosphere. It is a binary classification problem where positive cases (`g` for good) show evidence of some type of structure in the ionosphere and negative cases (`b` for bad) do not. It is a good dataset for practicing with neural networks because all of the inputs are small numerical values of the same scale. There are 34 attributes and 351 observations.

State-of-the-art results on this dataset achieve an accuracy of approximately 94% to 98% accuracy using 10-fold cross-validation. The dataset is available within the code bundle provided with this book. Alternatively, you can download it directly from the UCI Machine Learning repository[1]. Place the data file in your working directory with the filename `ionosphere.csv`. You can learn more about the ionosphere dataset on the UCI Machine Learning Repository website.

A small neural network model is constructed with a single hidden layer with 34 neurons, using the rectifier activation function. The output layer has a single neuron and uses the sigmoid activation function in order to output probability-like values. The learning rate for stochastic gradient descent has been set to a higher value of 0.1. The model is trained for 50 epochs, and the decay argument has been set to 0.002, calculated as $\frac{0.1}{50}$. Additionally, it can be a good idea to use momentum when using an adaptive learning rate. In this case, we use a momentum value of 0.8. The complete example is listed below.

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/ionosphere.csv

```
# Time Based Learning Rate Decay
from pandas import read_csv
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from sklearn.preprocessing import LabelEncoder
# load dataset
dataframe = read_csv("ionosphere.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:34].astype(float)
Y = dataset[:,34]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
Y = encoder.transform(Y)
# create model
model = Sequential()
model.add(Dense(34, input_shape=(34,), activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
epochs = 50
learning_rate = 0.1
decay_rate = learning_rate / epochs
momentum = 0.8
sgd = SGD(learning_rate=learning_rate, momentum=momentum, decay=decay_rate,
          nesterov=False)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=epochs, batch_size=28, verbose=2)
```

Listing 21.1: Example of time-based learning rate decay

The model is trained on 67% of the dataset and evaluated using a 33% validation dataset. Running the example shows a classification accuracy of 99.14%. This is higher than the baseline of 95.69% without the learning rate decay or momentum.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
...
Epoch 45/50
0s - loss: 0.0622 - acc: 0.9830 - val_loss: 0.0929 - val_acc: 0.9914
Epoch 46/50
0s - loss: 0.0695 - acc: 0.9830 - val_loss: 0.0693 - val_acc: 0.9828
Epoch 47/50
0s - loss: 0.0669 - acc: 0.9872 - val_loss: 0.0616 - val_acc: 0.9828
Epoch 48/50
0s - loss: 0.0632 - acc: 0.9830 - val_loss: 0.0824 - val_acc: 0.9914
Epoch 49/50
0s - loss: 0.0590 - acc: 0.9830 - val_loss: 0.0772 - val_acc: 0.9828
```

```
Epoch 50/50
0s - loss: 0.0592 - acc: 0.9872 - val_loss: 0.0639 - val_acc: 0.9828
```

Output 21.2: Sample output of time-based learning rate decay

21.3 Drop-Based Learning Rate Schedule

Another popular learning rate schedule used with deep learning models is systematically dropping the learning rate at specific times during training. Often this method is implemented by dropping the learning rate by half every fixed number of epochs. For example, we may have an initial learning rate of 0.1 and drop it by a factor of 0.5 every ten epochs. The first ten epochs of training would use a value of 0.1, and in the next ten epochs, a learning rate of 0.05 would be used, and so on. If we plot out the learning rates for this example out to 100 epochs, you get the graph below showing the learning rate (y-axis) versus epoch (x-axis).

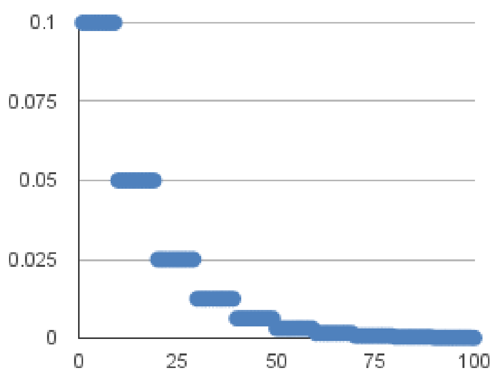


Figure 21.2: Drop-based learning rate schedule

You can implement this in Keras using the `LearningRateScheduler` callback when fitting the model. The `LearningRateScheduler` callback allows you to define a function to call that takes the epoch number as an argument and returns the learning rate to use in stochastic gradient descent. When used, the learning rate specified by stochastic gradient descent is ignored. In the code below, we use the same example as before of a single hidden layer network on the Ionosphere dataset. A new `step_decay()` function is defined that implements the equation:

$$\text{LearningRate} = \text{InitialLearningRate} \times \text{DropRate}^{\text{floor}(\frac{1+\text{Epoch}}{\text{EpochDrop}})}$$

Here, the InitialLearningRate is the initial learning rate (such as 0.1), the DropRate is the amount that the learning rate is modified each time it is changed (such as 0.5), Epoch is the current epoch number, and EpochDrop is how often to change the learning rate (such as 10). Notice that the learning rate in the SGD class is set to 0 to clearly indicate that it is not used. Nevertheless, you can set a momentum term in SGD if you want to use momentum with this learning rate schedule.

```
# Drop-Based Learning Rate Decay
from pandas import read_csv
import math
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.callbacks import LearningRateScheduler

# learning rate schedule
def step_decay(epoch):
    initial_lrate = 0.1
    drop = 0.5
    epochs_drop = 10.0
    lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
    return lrate

# load dataset
dataframe = read_csv("ionosphere.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:34].astype(float)
Y = dataset[:,34]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
Y = encoder.transform(Y)
# create model
model = Sequential()
model.add(Dense(34, input_shape=(34,), activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
sgd = SGD(learning_rate=0.0, momentum=0.9)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
# learning schedule callback
lrate = LearningRateScheduler(step_decay)
callbacks_list = [lrate]
# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=50, batch_size=28,
          callbacks=callbacks_list, verbose=2)
```

Listing 21.2: Example of drop-based learning rate decay

Running the example results in a classification accuracy of 99.14% on the validation dataset, again an improvement over the baseline for the model of this dataset.

```
...
Epoch 45/50
0s - loss: 0.0546 - acc: 0.9830 - val_loss: 0.0634 - val_acc: 0.9914
Epoch 46/50
0s - loss: 0.0544 - acc: 0.9872 - val_loss: 0.0638 - val_acc: 0.9914
Epoch 47/50
0s - loss: 0.0553 - acc: 0.9872 - val_loss: 0.0696 - val_acc: 0.9914
Epoch 48/50
```

```
0s - loss: 0.0537 - acc: 0.9872 - val_loss: 0.0675 - val_acc: 0.9914
Epoch 49/50
0s - loss: 0.0537 - acc: 0.9872 - val_loss: 0.0636 - val_acc: 0.9914
Epoch 50/50
0s - loss: 0.0534 - acc: 0.9872 - val_loss: 0.0679 - val_acc: 0.9914
```

Output 21.3: Sample output of drop-based learning rate decay

21.4 Tips for Using Learning Rate Schedules

This section lists some tips and tricks to consider when using learning rate schedules with neural networks.

- ▷ **Increase the initial learning rate**. Because the learning rate will decrease, start with a larger value to decrease from. A larger learning rate will result in much larger changes to the weights, at least in the beginning, allowing you to benefit from fine-tuning later.
- ▷ **Use a larger momentum**. Larger momentum value will make the update carry on to later steps. It helps the optimization algorithm continue to make updates in the right direction when your learning rate shrinks to small values.
- ▷ **Experiment with different schedules**. It will not be clear which learning rate schedule to use, so try a few with different configuration options and see what works best on your problem. Also, try schedules that change exponentially and even schedules that respond to the accuracy of your model on the training or test datasets.

21.5 Further Readings

This section provides more resources on the topic if you are looking to go deeper.

Articles

Ionosphere Data Set. UCI Machine Learning Repository.
https://archive.ics.uci.edu/ml/datasets/Ionosphere

21.6 Summary

In this chapter, you discovered learning rate schedules for training neural network models. You learned:

- ▷ The benefits of using learning rate schedules during training to lift model performance
- ▷ How to configure and use a time-based learning rate schedule in Keras
- ▷ How to develop your own drop-based learning rate schedule in Keras

In the next chapter, you will learn about different ways to feed training data to a Keras model.

22 Introduction to tf.data API

When you build and train a Keras deep learning model, you can provide the training data in several different ways. Presenting the data as a NumPy array or a TensorFlow tensor is common. Another way is to make a Python generator function and let the training loop read data from it. Yet another way of providing data is to use `tf.data` dataset.

In this chapter, you will see how you can use the `tf.data` dataset for a Keras model. After finishing this chapter, you will learn:

- ▷ How to create and use the `tf.data` dataset
- ▷ The benefit of doing so compared to a generator function

Let's get started.

Overview

This chapter is divided into four sections; they are:

- ▷ Training a Keras Model with NumPy Array and Generator Function
- ▷ Creating a Dataset using `tf.data`
- ▷ Creating a Dataest from Generator Function
- ▷ Data with Prefetch

22.1 Training a Keras Model with NumPy Array and Generator Function

Before you see how the `tf.data` API works, let's review how you usually train a Keras model.

First, you need a dataset. An example is the Fashion-MNIST dataset that comes with the Keras API. This dataset has 60,000 training samples and 10,000 test samples of 28×28 pixels in grayscale, and the corresponding classification label is encoded with integers 0 to 9. The dataset is a NumPy array. Then you can build a Keras model for classification, and with the model's `fit()` function, you provide the NumPy array as data.

The complete code is as follows:

```
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets.fashion_mnist import load_data
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Sequential

(train_image, train_label), (test_image, test_label) = load_data()
print(train_image.shape)
print(train_label.shape)
print(test_image.shape)
print(test_label.shape)

model = Sequential([
    Flatten(input_shape=(28,28)),
    Dense(100, activation="relu"),
    Dense(100, activation="relu"),
    Dense(10, activation="sigmoid")
])
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics="sparse_categorical_accuracy")
history = model.fit(train_image, train_label,
                    batch_size=32, epochs=50,
                    validation_data=(test_image, test_label), verbose=0)
print(model.evaluate(test_image, test_label))

plt.plot(history.history['val_sparse_categorical_accuracy'])
plt.show()
```

Listing 22.1: Training a neural network with dataset in NumPy arrays

Running this code will plot the validation accuracy over the 50 epochs you trained your model:

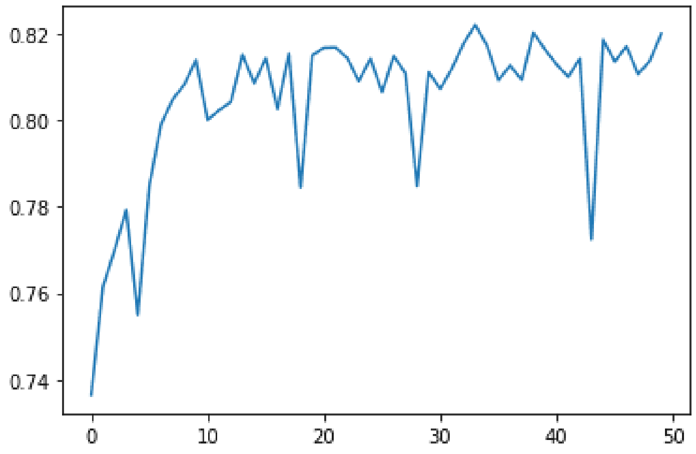


Figure 22.1: Validation accuracy achieved during 50 epochs of training

And also, print out the matrices' shape and the evaluation score:

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
313/313 [==============================] - 0s 392us/step - loss: 0.5114 - sparse_categoric
[0.5113903284072876, 0.8446000218391418]
```

Output 22.1: Screen output of training with dataset in NumPy array

The other way of training the same network is to provide the data from a Python generator function instead of a NumPy array. A generator function is the one with a `yield` statement to emit data while the function runs in parallel to the data consumer. A generator of the Fashion-MNIST dataset can be created as follows:

```python
def batch_generator(image, label, batchsize):
    N = len(image)
    i = 0
    while True:
        yield image[i:i+batchsize], label[i:i+batchsize]
        i = i + batchsize
        if i + batchsize > N:
            i = 0
```

Listing 22.2: A batch generator for training a Keras model

This function is supposed to be called with the syntax

```python
batch_generator(train_image, train_label, 32)
```

It will scan the input arrays in batches indefinitely. Once it reaches the end of the array, it will restart from the beginning.

Training a Keras model with a generator is similar to using the `fit()` function:

```python
history = model.fit(batch_generator(train_image, train_label, 32),
                    steps_per_epoch=len(train_image)//32,
                    epochs=50, validation_data=(test_image, test_label), verbose=0)
```

Listing 22.3: Training a Keras model with a generator

Instead of providing the data and label, you just need to provide the generator as it will give out both. When data are presented as a NumPy array, you can tell how many samples there are by looking at the length of the array. Keras can complete one epoch when the entire dataset is used once. However, your generator function will emit batches indefinitely, so you need to tell it when an epoch is ended, using the `steps_per_epoch` argument to the `fit()` function.

In the above code, the validation data was provided as a NumPy array, but you can also use a generator instead and specify the `validation_steps` argument.

The following is the complete code using a generator function, in which the output is the same as the previous example:

```python
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets.fashion_mnist import load_data
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Sequential

model = Sequential([
    Flatten(input_shape=(28,28)),
    Dense(100, activation="relu"),
    Dense(100, activation="relu"),
    Dense(10, activation="sigmoid")
])

def batch_generator(image, label, batchsize):
    N = len(image)
    i = 0
    while True:
        yield image[i:i+batchsize], label[i:i+batchsize]
        i = i + batchsize
        if i + batchsize > N:
            i = 0

model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics="sparse_categorical_accuracy")
history = model.fit(batch_generator(train_image, train_label, 32),
                    steps_per_epoch=len(train_image)//32,
                    epochs=50, validation_data=(test_image, test_label), verbose=2)
print(model.evaluate(test_image, test_label))

plt.plot(history.history['val_sparse_categorical_accuracy'])
plt.show()
```

Listing 22.4: Complete example of using a generator function to train a Keras model

22.2 Creating a Dataset Using tf.data

Given that you have the Fashion-MNIST data loaded, you can convert it into a `tf.data` dataset, like the following:

```python
...
dataset = tf.data.Dataset.from_tensor_slices((train_image, train_label))
print(dataset.element_spec)
```

Listing 22.5: Creating a `tf.data` dataset

This prints the dataset's spec as follows:

```
(TensorSpec(shape=(28, 28), dtype=tf.uint8, name=None),
 TensorSpec(shape=(), dtype=tf.uint8, name=None))
```

Output 22.2: Data spec of a `tf.data` dataset

You can see the data is a tuple (as a tuple was passed as an argument to the `from_tensor_slices()` function), whereas the first element is in the shape `(28,28)` while the second element is a scalar. Both elements are stored as 8-bit unsigned integers.

If you do not present the data as a tuple of two NumPy arrays when you create the dataset, you can also do it later. The following creates the same dataset but first creates the dataset for the image data and the label separately before combining them:

```
...
train_image_data = tf.data.Dataset.from_tensor_slices(train_image)
train_label_data = tf.data.Dataset.from_tensor_slices(train_label)
dataset = tf.data.Dataset.zip((train_image_data, train_label_data))
print(dataset.element_spec)
```

Listing 22.6: Creating a dataset by combining two other datasets

This will print the same spec:

```
(TensorSpec(shape=(28, 28), dtype=tf.uint8, name=None),
 TensorSpec(shape=(), dtype=tf.uint8, name=None))
```

Output 22.3: Data spec of a `tf.data` dataset

The `zip()` function in the dataset is like the `zip()` function in Python because it matches data one by one from multiple datasets into a tuple.

One benefit of using the `tf.data` dataset is the flexibility in handling the data. Below is the complete code on how you can train a Keras model using a dataset in which the batch size is set to the dataset:

```
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets.fashion_mnist import load_data
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Sequential

(train_image, train_label), (test_image, test_label) = load_data()
dataset = tf.data.Dataset.from_tensor_slices((train_image, train_label))

model = Sequential([
    Flatten(input_shape=(28,28)),
    Dense(100, activation="relu"),
    Dense(100, activation="relu"),
    Dense(10, activation="sigmoid")
])
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics="sparse_categorical_accuracy")
history = model.fit(dataset.batch(32),
                    epochs=50, validation_data=(test_image, test_label), verbose=2)
print(model.evaluate(test_image, test_label))

plt.plot(history.history['val_sparse_categorical_accuracy'])
plt.show()
```

Listing 22.7: Training a neural network with `tf.data` dataset

This is the simplest use case of using a dataset. If you dive deeper, you can see that a dataset is just an iterator. Therefore, you can print out each sample in a dataset using the following:

```
for image, label in dataset:
    print(image)  # array of shape (28,28) in tf.Tensor
    print(label)  # integer label in tf.Tensor
```

Listing 22.8: Iteratively extracting samples from a dataset

The dataset has many functions built in. The `batch()` used before is one of them. If you create batches from dataset and print them, you have the following:

```
for image, label in dataset.batch(32):
    print(image)  # array of shape (32,28,28) in tf.Tensor
    print(label)  # array of shape (32,) in tf.Tensor
```

Listing 22.9: Iteratively extracting batchs from a dataset

Here, each item from a batch is not a sample but a batch of samples. You also have functions such as `map()`, `filter()`, and `reduce()` for sequence transformation, or `concatendate()` and `interleave()` for combining with another dataset. There are also `repeat()`, `take()`, `take_while()`, and `skip()` like our familiar counterpart from Python's `itertools` module. A full list of the functions can be found in the API documentation.

22.3 Creating a Dataset from Generator Function

So far, you saw how a dataset could be used in place of a NumPy array in training a Keras model. Indeed, a dataset can also be created out of a generator function. But instead of a generator function that generates a *batch*, as you saw in one of the examples above, you now make a generator function that generates one sample at a time. The following is the function:

```
import numpy as np
import tensorflow as tf

def shuffle_generator(image, label, seed):
    idx = np.arange(len(image))
    np.random.default_rng(seed).shuffle(idx)
    for i in idx:
        yield image[i], label[i]

dataset = tf.data.Dataset.from_generator(
    shuffle_generator,
    args=[train_image, train_label, 42],
    output_signature=(
        tf.TensorSpec(shape=(28,28), dtype=tf.uint8),
        tf.TensorSpec(shape=(), dtype=tf.uint8)))
print(dataset.element_spec)
```

Listing 22.10: Creating a dataset from a generator function

This function randomizes the input array by shuffling the index vector. Then it generates one sample at a time. Unlike the previous example, this generator will end when the samples from the array are exhausted.

You can create a dataset from the function using `from_generator()`. You need to provide the name of the generator function (instead of an instantiated generator) and also the output signature of the dataset. This is required because the `tf.data.Dataset` API cannot infer the dataset spec before the generator is consumed.

Running the above code will print the same spec as before:

```
(TensorSpec(shape=(28, 28), dtype=tf.uint8, name=None),
 TensorSpec(shape=(), dtype=tf.uint8, name=None))
```

Output 22.4: Data spec of a dataset created with the generator function

Such a dataset is functionally equivalent to the dataset that you created previously. Hence you can use it for training as before. The following is the complete code:

```python
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets.fashion_mnist import load_data
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Sequential

(train_image, train_label), (test_image, test_label) = load_data()

def shuffle_generator(image, label, seed):
    idx = np.arange(len(image))
    np.random.default_rng(seed).shuffle(idx)
    for i in idx:
        yield image[i], label[i]

dataset = tf.data.Dataset.from_generator(
    shuffle_generator,
    args=[train_image, train_label, 42],
    output_signature=(
        tf.TensorSpec(shape=(28,28), dtype=tf.uint8),
        tf.TensorSpec(shape=(), dtype=tf.uint8)))

model = Sequential([
    Flatten(input_shape=(28,28)),
    Dense(100, activation="relu"),
    Dense(100, activation="relu"),
    Dense(10, activation="sigmoid")
])
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics="sparse_categorical_accuracy")
history = model.fit(dataset.batch(32),
                    epochs=50, validation_data=(test_image, test_label), verbose=2)
print(model.evaluate(test_image, test_label))
```

```
plt.plot(history.history['val_sparse_categorical_accuracy'])
plt.show()
```

Listing 22.11: Training a neural network with `tf.data` dataset created using a generator

22.4 Dataset with Prefetch

The real benefit of using a dataset is to use `prefetch()`.

Using a NumPy array for training is probably the best in performance. However, this means you need to load all data into memory. Using a generator function for training allows you to prepare one batch at a time, in which the data can be loaded from disk on demand, for example. Nonetheless, using a generator function to train a Keras model means either the training loop or the generator function is running at any time. If you work on a computer vision problem which the data are images on disk, and you need to do some image preprocessing before applying to your model, you want to run gradient descent on one batch of image while preparing the next batch. It is not easy to make the generator function and Keras' training loop run in parallel.

Dataset is the API that allows the generator and the training loop to run in parallel. If you have a generator that is computationally expensive (e.g., doing image augmentation in realtime), you can create a dataset from such a generator function and then use it with `prefetch()`, as follows:

```
...
history = model.fit(dataset.batch(32).prefetch(3),
                    epochs=50,
                    validation_data=(test_image, test_label),
                    verbose=0)
```

Listing 22.12: Using `prefetch()` with a dataset

The number argument to `prefetch()` is the size of the buffer. Here, the dataset is asked to keep three batches in memory ready for the training loop to consume. Whenever a batch is consumed, the dataset API will resume the generator function to refill the buffer asynchronously in the background. Therefore, you can allow the training loop and the data preparation algorithm inside the generator function to run in parallel.

It's worth mentioning that, in the previous section, you created a shuffling generator for the dataset API. Indeed the dataset API also has a `shuffle()` function to do the same, but you may not want to use it unless the dataset is small enough to fit in memory.

The `shuffle()` function, same as `prefetch()`, takes a buffer size argument. The shuffle algorithm will fill the buffer with the dataset and draw one element randomly from it. The consumed element will be replaced with the next element from the dataset. Hence you need the buffer as large as the dataset itself to make a truly random shuffle. This limitation is demonstrated with the following snippet:

```python
import tensorflow as tf
import numpy as np

n_dataset = tf.data.Dataset.from_tensor_slices(np.arange(10000))
shuffled = []
for n in n_dataset.shuffle(10).take(20):
    shuffled.append(n.numpy())
print(shuffled)
```

Listing 22.13: Using `shuffle()` with a small buffer

The output from the above looks like the following:

```
[7, 4, 1, 10, 13, 2, 6, 0, 3, 18, 8, 17, 15, 20, 22, 9, 12, 25, 21, 5]
```

Output 22.5: Data from a dataset shuffled with small buffer

Here you can see the numbers are shuffled around its neighborhood, and you never see large numbers from its output.

22.5 Further Reading

More about the `tf.data` dataset can be found from its API documentation:

`tf.data.Datasets`. TensorFlow API.
    https://www.tensorflow.org/datasets

22.6 Summary

In this chapter, you have seen how you can use the `tf.data` dataset and how it can be used in training a Keras model. Specifically, you learned:

▷ How to train a model using data from a NumPy array, a generator, and a dataset

▷ How to create a dataset using a NumPy array or a generator function

▷ How to use prefetch with a dataset to make the generator and training loop run in parallel

Start from the next chapter, you will learn about a different kind of neural networks that applies to images.

IV
Convolutional Neural Networks

23 Crash Course in Convolutional Neural Networks

Convolutional neural networks are a powerful artificial neural network technique. These networks preserve the spatial structure of the problem and were developed for object recognition tasks such as handwritten digit recognition. They are popular because people can achieve state-of-the-art results on challenging computer vision and natural language processing tasks. In this chapter, you will discover convolutional neural networks for deep learning, also called ConvNets or CNNs. After completing this crash course, you will know:

- ▷ The building blocks used in CNNs, such as convolutional layers and pool layers
- ▷ How the building blocks fit together with a short worked example
- ▷ Best practices for configuring CNNs on your object recognition tasks
- ▷ References for state-of-the-art networks applied to complex machine learning problems

Let's get started.

Overview

This chapter is divded into four sections; they are:

- ▷ The Case for Convolutional Neural Networks
- ▷ Building Blocks of Convolutional Neural Networks
- ▷ Convolutional Layers
- ▷ Pooling Layers
- ▷ Fully Connected Layers
- ▷ Worked Example of a Convolutional Neural Network
- ▷ Convolutional Neural Networks Best Practices

23.1 The Case for Convolutional Neural Networks

Given a dataset of grayscale images with the standardized size of 32×32 pixels each, a traditional feedforward neural network would require 1024 input weights (plus one bias). This is fair enough, but the flattening of the image matrix of pixels to a long vector of pixel values loses all the spatial structure in the image. Unless all the images are perfectly resized, the neural network will have great difficulty with the problem.

Convolutional neural networks use fewer weights. They also expect and preserve the spatial relationship between pixels by learning internal feature representations using small squares of input data. Features are learned and used across the whole image, allowing the objects in the images to be shifted or translated in the scene but still detectable by the network. This is why the network is so useful for object recognition in photographs, picking out digits, faces, objects, and so on with varying orientations. In summary, below are some of the benefits of using convolutional neural networks:

- ▷ They use fewer parameters (weights) to learn than a fully connected network
- ▷ They are designed to be invariant to object position and distortion in the scene
- ▷ They automatically learn and generalize features from the input domain

23.2 Building Blocks of Convolutional Neural Networks

There are three types of layers in a convolutional neural network:

1. Convolutional Layers
2. Pooling Layers
3. Fully-Connected Layers

23.3 Convolutional Layers

Convolutional layers are comprised of filters and feature maps.

Filters

Filters are essentially the *neurons* of the layer. They take weighted inputs and output a value. The input size is a fixed square called a patch or a *receptive field*. If the convolutional layer is an input layer, the input patch will be the pixel values. If deeper in the network architecture, the convolutional layer will take input from a feature map from the previous layer.

Feature Maps

A feature map is the output of one filter applied to the previous layer. A given filter is drawn across the entire previous layer and moved one pixel at a time. Each position results in the activation of the neuron, and the output is collected in the feature map. You can see that if the receptive field is moved one pixel from activation to activation, then the field will overlap with the previous activation by (field width−1) input values.

The distance that filter is moved across the input from the previous layer for each activation is referred to as the stride. If the size of the previous layer is not cleanly divisible by the size of the filter's receptive field and the size of the stride, then it is possible for the receptive field to attempt to read off the edge of the input feature map. In this case, techniques like zero padding can be used to invent mock inputs with zero values for the receptive field to read.

23.4 Pooling Layers

The pooling layers downsample the previous layer's feature map. Pooling layers follow a sequence of one or more convolutional layers and are intended to consolidate the features learned and expressed in the previous layer's feature map. As such, pooling may be considered a technique to compress or generalize feature representations and generally reduce the overfitting of the training data by the model.

They, too, have a receptive field, often much smaller than the convolutional layer. Also, the stride or number of inputs that the receptive field is moved for each activation is often equal to the size of the receptive field to avoid any overlap. Pooling layers are often very simple, taking the average (average pooling) or the maximum (max pooling) of the input values in order to create its own feature map.

23.5 Fully Connected Layers

Fully connected layers are the normal flat feedforward neural network layer. These layers may have a nonlinear activation function or a softmax activation in order to output probabilities of class predictions. Fully connected layers are used at the end of the network after feature extraction and consolidation have been performed by the convolutional and pooling layers. They are used to create final nonlinear combinations of features and for making predictions by the network.

23.6 Worked Example of a Convolutional Neural Network

You now know about convolutional, pooling, and fully connected layers. Let's make this more concrete by working through how these three layers may be connected together.

Image Input Data

Let's assume you have a dataset of grayscale images. Each image has the same size of 32 pixels wide and 32 pixels high, and pixel values are between 0 and 255, e.g., a matrix of $32 \times 32 \times 1$ or 1024 pixel values. Image input data is expressed as a 3-dimensional matrix of width $\times$ height $\times$ channels. If you were using color images in the example, you would have three channels for the red, green, and blue pixel values, e.g., $32 \times 32 \times 3$.

Convolutional Layer

Define a convolutional layer with ten filters, and a receptive field 5 pixels wide and 5 pixels high, and a stride length of 1. Because each filter can only get input from (i.e., *see*) 5×5 or 25 pixels at a time, you can calculate that each will require $25 + 1$ input weights (plus 1 for the bias input). Dragging the 5×5 receptive field across the input image data with a stride width of 1 will result in a feature map of 28×28 output values or 784 distinct activations per image.

You have ten filters, so ten different 28×28 feature maps or 7,840 outputs will be created for one image. Finally, you know you have 26 inputs per filter, ten filters, and 28×28 output values to calculate per filter. Therefore, you have a total of $26 \times 10 \times 28 \times 28$ or 203,840 *connections* in your convolutional layer if you want to phrase it using traditional neural network nomenclature. Convolutional layers also make use of a nonlinear transfer function as part of the activation, and the rectifier activation function is the popular default to use.

Pooling Layer

You can define a pooling layer with a receptive field with a width of 2 inputs and a height of 2 inputs. You can also use a stride of 2 to ensure that there is no overlap. This results in feature maps that are one-half the size of the input feature maps, from ten different 28×28 feature maps as input to ten different 14×14 feature maps as output. You will use a `max()` operation for each receptive field so that the activation is the maximum input value.

Fully Connected Layer

Finally, you can flatten out the square feature maps into a traditional flat, fully connected layer. You can define the fully connected layer with 200 hidden neurons, each with $10 \times 14 \times 14$ input connections, or $1,960 + 1$ weights per neuron. That is a total of 392,200 connections and weights to learn in this layer. You can use a sigmoid or softmax transfer function to output probabilities of class values directly.

23.7 Convolutional Neural Networks Best Practices

Now that you know about the building blocks for a convolutional neural network and how the layers hang together, you can review some best practices to consider when applying them.

- **Input Receptive Field Dimensions**: The default is 2D for images but could be 1D for words in a sentence or 3D for a video that adds a time dimension.
- **Receptive Field Size**: The patch should be as small as possible but large enough to *see* features in the input data. It is common to use 3×3 on small images and 5×5 or 7×7 and more on larger image sizes.
- **Stride Width**: Use the default stride of 1. It is easy to understand, and you don't need padding to handle the receptive field falling off the edge of your images. This could be increased to 2 or larger for larger images.
- **Number of Filters**: Filters are the feature detectors. Generally, fewer filters are used at the input layer, and increasingly more filters are used at deeper layers.
- **Padding**: Set to zero and called zero padding when reading non-input data. This is useful when you cannot or do not want to standardize input image sizes or when you want to use receptive field and stride sizes that do not neatly divide up the input image size.

- ▷ **Pooling**: Pooling is a destructive or generalization process to reduce overfitting. The receptive field size is almost always set to 2×2 with a stride of 2 to discard 75% of the activations from the output of the previous layer.
- ▷ **Data Preparation**: Consider standardizing input data, both the dimensions of the images and pixel values.
- ▷ **Pattern Architecture**: It is common to pattern the layers in your network architecture. This might be one, two, or some number of convolutional layers followed by a pooling layer. This structure can then be repeated one or more times. Finally, fully connected layers are often only used at the output end and may be stacked one, two, or more deep.
- ▷ **Dropout**: CNNs have a habit of overfitting, even with pooling layers. Dropout should be used, such as between fully connected layers and perhaps after pooling layers.

23.8 Further Readings

You have only scratched the surface on convolutional neural networks. The field is moving very fast, and new and interesting architectures and techniques are being discussed and used all the time.

If you are looking for a deeper understanding of the technique, take a look at LeCun, Bottou, et al.'s seminal paper titled "Gradient-Based Learning Applied to Document Recognition". In it, they introduce LeNet applied to handwritten digit recognition and carefully explain the layers and how the network is connected.

There are a lot of tutorials and discussions of CNNs around the web. A few chosen examples are listed below. Personally, I find the explanatory pictures in the posts useful only after understanding how the network hangs together. Many of the explanations are confusing and I recommend you to read LeCun's paper if in doubt.

Articles

Yann LeCun, Leon Bottou, et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE*. Nov. 1998.
http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

Yann LeCun. *LeNet-5 convolutional neural networks.*
http://yann.lecun.com/exdb/lenet/

Convolutional Neural Networks. Stanford Course CS231n. Convolutional Neural Networks for Visual Recognition.
https://cs231n.github.io/convolutional-networks/

Yann LeCun, Koray Kavukcuoglu, and Clement Farabet. "Convolutional Networks and Applications in Vision". In: *Proceedings of IEEE International Symposium on Circuits and Systems*. Paris, France, 2010.
http://yann.lecun.com/exdb/publis/pdf/lecun-iscas-10.pdf

Michael Nielsen. Chapter 6, "Deep Learning". In: *Neural Networks abd Deep Learning.* Determination Press, 2015.
http://neuralnetworksanddeeplearning.com/chap6.html

Andrea Vedaldi and Andrew Zisserman. *VGG Convolutional Neural Networks Practical.* Oxford Visual Geometry Group, 2017.
http://www.robots.ox.ac.uk/~vgg/practicals/cnn/

Denny Britz. *Understanding Convolutional Neural Networks for NLP*. Nov. 2015.
https://www.kdnuggets.com/2015/11/understanding-convolutional-neural-networks-nlp.html

23.9 Summary

In this chapter, you discovered convolutional neural networks. You learned about:

- ▷ Why CNNs are needed to preserve spatial structure in your input data and the benefits they provide
- ▷ The building blocks of CNN include convolutional, pooling, and fully connected layers
- ▷ How the layers in a CNN hang together
- ▷ Best practices when applying a CNN to your own problems

In the next chapter, you will look closer to the hyperparameters of a convolutional layer.

24 Understanding the Design of a Convolutional Neural Network

Convolutional neural networks have been found successful in computer vision applications. Various network architectures are proposed and they are neither magical nor hard to understand. In this chapter, you will make sense of the operation of convolutional layers and their role in a larger convolutional neural network. After finishing this chapter, you will learn:

▷ How convolutional layers extract features from image

▷ How different convolutional layers can stack up to build a neural network

Let's get started.

Overview

This chapter is divided into three sections; they are:

▷ An Example Network

▷ Showing the Feature Maps

▷ Effect of the Convolutional Layers

24.1 An Example Network

The following is a program to do image classification on the CIFAR-10 dataset:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D, Flatten, Dense
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.datasets.cifar10 import load_data


(X_train, y_train), (X_test, y_test) = load_data()
```

```
# rescale image
X_train_scaled = X_train / 255.0
X_test_scaled = X_test / 255.0

model = Sequential([
    Conv2D(32, (3,3), input_shape=(32, 32, 3), padding="same",
           activation="relu", kernel_constraint=MaxNorm(3)),
    Dropout(0.3),
    Conv2D(32, (3,3), padding="same",
           activation="relu", kernel_constraint=MaxNorm(3)),
    MaxPooling2D(),
    Flatten(),
    Dense(512, activation="relu", kernel_constraint=MaxNorm(3)),
    Dropout(0.5),
    Dense(10, activation="sigmoid")
])

model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics="sparse_categorical_accuracy")

model.fit(X_train_scaled, y_train, epochs=25, batch_size=32,
          validation_data=(X_test_scaled, y_test))
```

Listing 24.1: Image classification on CIFAR-10 dataset

This network should be able to achieve around 70% accuracy in classification. The images are in 32×32 pixels in RGB color. They are in 10 different classes, and the labels are integers from 0 to 9.

You can print the network using Keras' `summary()` function:

```
...
model.summary()
```

Listing 24.2: Print a network model

In this network, the following will be shown on the screen:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

 dropout (Dropout)           (None, 32, 32, 32)        0

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d (MaxPooling2D)  (None, 16, 16, 32)      0

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 512)               4194816
 dropout_1 (Dropout)          (None, 512)              0

 dense_1 (Dense)              (None, 10)               5130

=================================================================
Total params: 4,210,090
Trainable params: 4,210,090
Non-trainable params: 0
_________________________________________________________________
```

Output 24.1: The network model we created in Listing 24.1

It is typical in a network for image classification to be comprised of convolutional layers at an early stage, with dropout and pooling layers interleaved. Then, at a later stage, the output from convolutional layers is flattened and processed by some fully connected layers.

24.2 Showing the Feature Maps

In the above network, there are two convolutional layers (`Conv2D`). The first layer is defined as follows:

```
Conv2D(32, (3,3), input_shape=(32, 32, 3), padding="same", activation="relu",
       kernel_constraint=MaxNorm(3))
```

Listing 24.3: The first layer in our network model

This means the convolutional layer will have a 3×3 kernel and apply on an input image of 32×32 pixels and three channels (the RGB colors). therefore, the output of this layer will be 32 channels.

In order to make sense of the convolutional layer, you can check out its kernel. The variable `model` holds the network, and you can find the kernel of the first convolutional layer with the following:

```
...
print(model.layers[0].kernel)
```

Listing 24.4: Show the kernel of the first layer in our model

This prints:

```
<tf.Variable 'conv2d/kernel:0' shape=(3, 3, 3, 32) dtype=float32, numpy=
array([[[[-2.30068922e-01,  1.41024575e-01, -1.93124503e-01,
          -2.03153938e-01,  7.71819279e-02,  4.81446862e-01,
          -1.11971676e-01, -1.75487325e-01, -4.01797555e-02,
          ...
           4.64215249e-01,  4.10646647e-02,  4.99733612e-02,
          -5.22711873e-02, -9.20209661e-03, -1.16479330e-01,
           9.25614685e-02, -4.43541892e-02]]]], dtype=float32)>
```

Output 24.2: The kernel in the first layer of our model

You can tell that `model.layers[0]` is the correct layer by comparing the name `conv2d` from the above output to the output of `model.summary()`. This layer has a kernel of shape `(3, 3, 3, 32)`, which are the height, width, input channels, and output feature maps, respectively.

Assume the kernel is a NumPy array `k`. A convolutional layer will take its kernel `k[:, :, 0, n]` (a 3×3 array) and apply on the first channel of the image. Then apply `k[:, :, 1, n]` on the second channel of the image, and so on. Afterward, the result of the convolution on all the channels is added up to become the feature map `n` of the output, where `n`, in this case, will run from 0 to 31 for the 32 output feature maps.

In Keras, you can extract the output of each layer using an extractor model. In the following, you will create a batch with one input image and send it to the network. Then look at the feature maps of the first convolutional layer:

```
...
# Extract output from each layer
extractor = tf.keras.Model(inputs=model.inputs,
                           outputs=[layer.output for layer in model.layers])
features = extractor(np.expand_dims(X_train_scaled[7], 0))

# Show the 32 feature maps from the first layer
l0_features = features[0].numpy()[0]

fig, ax = plt.subplots(4, 8, sharex=True, sharey=True, figsize=(16,8))
for i in range(0, 32):
    row, col = i//8, i%8
    ax[row][col].imshow(l0_features[..., i])

plt.show()
```

Listing 24.5: Showing the feature map from the first convolutional layer

The above code will print the feature maps like the following:

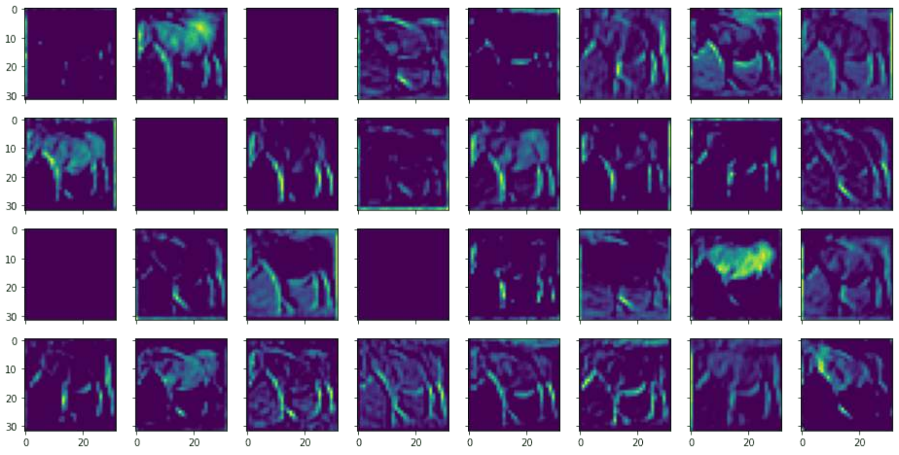


Figure 24.1: The feature map of the first layer

This corresponds to the following input image:

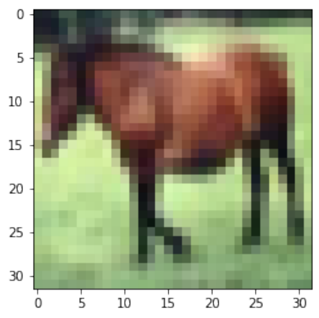


Figure 24.2: The input image to our network

You can see that they are called feature maps because they are highlighting certain features from the input image. A feature is identified using a small window (in this case, over a 3×3 pixels filter). The input image has three color channels. Each channel has a different filter applied, and their results are combined for an output feature.

You can similarly display the feature map from the output of the second convolutional layer as follows:

```
...
# Show the 32 feature maps from the third layer
l2_features = features[2].numpy()[0]

fig, ax = plt.subplots(4, 8, sharex=True, sharey=True, figsize=(16,8))
for i in range(0, 32):
    row, col = i//8, i%8
    ax[row][col].imshow(l2_features[..., i])

plt.show()
```

Listing 24.6: Showing the feature map from the second convolutional layer

This shows the following:

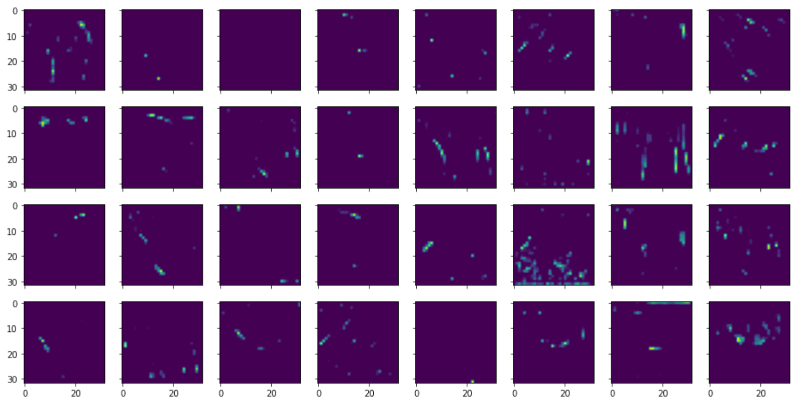


Figure 24.3: The feature map of the second layer

From the above, you can see that the features extracted are more abstract and less recognizable.

The following is the complete code to create, train, and visualize the feature maps from a trained network.

```python
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D, Flatten, Dense
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.datasets.cifar10 import load_data


(X_train, y_train), (X_test, y_test) = load_data()

# rescale image
X_train_scaled = X_train / 255.0
X_test_scaled = X_test / 255.0

model = Sequential([
    Conv2D(32, (3,3), input_shape=(32, 32, 3), padding="same",
           activation="relu", kernel_constraint=MaxNorm(3)),
    Dropout(0.3),
    Conv2D(32, (3,3), padding="same",
           activation="relu", kernel_constraint=MaxNorm(3)),
    MaxPooling2D(),
    Flatten(),
    Dense(512, activation="relu", kernel_constraint=MaxNorm(3)),
    Dropout(0.5),
    Dense(10, activation="sigmoid")
])

# Train the network with CIFAR10 dataset
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics="sparse_categorical_accuracy")

model.fit(X_train_scaled, y_train, epochs=25, batch_size=32,
          validation_data=(X_test_scaled, y_test))

# Visualize the input image
plt.imshow(X_train_scaled[7])
plt.show()

# Extract output from each layer
extractor = tf.keras.Model(inputs=model.inputs,
                           outputs=[layer.output for layer in model.layers])
features = extractor(np.expand_dims(X_train_scaled[7], 0))

# Show the 32 feature maps from the first layer
l0_features = features[0].numpy()[0]

fig, ax = plt.subplots(4, 8, sharex=True, sharey=True, figsize=(16,8))
for i in range(0, 32):
    row, col = i//8, i%8
    ax[row][col].imshow(l0_features[..., i])

plt.show()
```

```
# Show the 32 feature maps from the third layer
l2_features = features[2].numpy()[0]

fig, ax = plt.subplots(4, 8, sharex=True, sharey=True, figsize=(16,8))
for i in range(0, 32):
    row, col = i//8, i%8
    ax[row][col].imshow(l2_features[..., i])

plt.show()
```

Listing 24.7: Visualizing the feature map from a network

24.3 Effect of the Convolutional Layers

The most important hyperparameter to a convolutional layer is the size of the filter. Usually, it is in a square shape, and you can consider that as a *window* or *receptive field* to look at the input image. Therefore, the higher resolution of the image, then you can expect a larger filter.

On the other hand, a filter too large will blur the detailed features because all pixels from the receptive field through the filter will be combined into one pixel at the output feature map. Therefore, there is a trade-off for the appropriate size of the filter.

Stacking two convolutional layers (without any other layers in between) is equivalent to a single convolutional layer with a larger filter. But a typical design to use nowadays is two layers with small filters stacked together rather than one larger with a larger filter, as there are fewer parameters to train.

The exception would be a convolutional layer with a 1×1 filter. This is usually found as the beginning layer of a network. The purpose of such a convolutional layer is to combine the input channels into one rather than transforming the pixels. Conceptually, this can convert a color image into grayscale, but usually, you can use multiple ways of conversion to create more input channels than merely RGB for the network.

Also, note that in the above network, you are using `Conv2D` for a 2D filter. There is also a `Conv3D` layer for a 3D filter. The difference is whether you apply the filter separately for each channel or feature map or consider the input feature maps stacked up as a 3D array and apply a single filter to transform it altogether. Usually, the former is used as it is more reasonable to consider no particular order in which the feature maps should be stacked.

Besides the size of the receptive field, *padding* and *stride* are two other parameters of a convolutional layer. Padding describes how far the receptive field can go outside the border of the input image. Stride describes how big a step the receptive field would move on the input image to produce the layer's next output.

24.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Articles

Convolutional Neural Networks. Stanford Course CS231n. Convolutional Neural Networks for Visual Recognition.
`https://cs231n.github.io/convolutional-networks/`

Yann LeCun, Koray Kavukcuoglu, and Clement Farabet. "Convolutional Networks and Applications in Vision". In: *Proceedings of IEEE International Symposium on Circuits and Systems.* Paris, France, 2010.
`http://yann.lecun.com/exdb/publis/pdf/lecun-iscas-10.pdf`

Michael Nielsen. Chapter 6, "Deep Learning". In: *Neural Networks abd Deep Learning.* Determination Press, 2015.
`http://neuralnetworksanddeeplearning.com/chap6.html`

24.5 Summary

In this chapter, you have seen how to visualize the feature maps from a convolutional neural network and how it works to extract the feature maps.

Specifically, you learned:

▷ The structure of a typical convolutional neural networks

▷ What is the effect of the filter size to a convolutional layer

▷ What is the effect of stacking convolutional layers in a network

In the next chapter, you will see an example of using CNN to recognize images.

Project: Handwritten Digit Recognition 25

A popular demonstration of the capability of deep learning techniques is object recognition in image data. The "hello world" of object recognition for machine learning and deep learning is the MNIST dataset for handwritten digit recognition. In this project, you will discover how to develop a deep learning model to achieve near state-of-the-art performance on the MNIST handwritten digit recognition task in Python using the Keras deep learning library. After completing this chapter, you will know:

- ▷ How to load the MNIST dataset in Keras
- ▷ How to develop and evaluate a baseline neural network model for the MNIST problem
- ▷ How to implement and evaluate a simple Convolutional Neural Network for MNIST
- ▷ How to implement a close to state-of-the-art deep learning model for MNIST

Let's get started.

Note: You may want to speed up the computation for this chapter by using GPU rather than CPU hardware, such as the process described in Appendix C. This is a suggestion, not a requirement. The code will work just fine on the CPU.

25.1 Description of the MNIST Handwritten Digit Recognition Problem

The MNIST problem is a dataset developed by Yann LeCun, Corinna Cortes, and Christopher Burges for evaluating machine learning models on the handwritten digit classification problem. The dataset was constructed from a number of scanned document datasets available from the National Institute of Standards and Technology (NIST). This is where the name for the dataset comes from, the Modified NIST or MNIST dataset.

Images of digits were taken from a variety of scanned documents, normalized in size, and centered. This makes it an excellent dataset for evaluating models, allowing the developer to focus on machine learning with minimal data cleaning or preparation required. Each image is a 28×28-pixel square (784 pixels total). A standard split of the dataset is used to evaluate and compare models, where 60,000 images are used to train a model, and a separate set of 10,000 images are used to test it.

It is a digit recognition task. As such, there are ten digits (0 to 9) or ten classes to predict. Results are reported using prediction error, which is nothing more than the inverted classification accuracy. Excellent results achieve a prediction error of less than 1%. A state-of-the-art prediction error of approximately 0.2% can be achieved with large convolutional neural networks. There is a listing of the state-of-the-art results and links to the relevant papers on the MNIST and other datasets on Rodrigo Benenson's webpage.

25.2 Loading the MNIST Dataset in Keras

The Keras deep learning library provides a convenient method for loading the MNIST dataset. The dataset is downloaded automatically the first time this function is called and stored in your home directory in `~/.keras/datasets/mnist.npz` as an 11MB file. This is very handy for developing and testing deep learning models. To demonstrate how easy it is to load the MNIST dataset, first, write a little script to download and visualize the first four images in the training dataset.

```python
# Plot ad hoc mnist instances
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
# load (downloaded if needed) the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# plot 4 images as gray scale
plt.subplot(221)
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.subplot(222)
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
plt.subplot(223)
plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
plt.subplot(224)
plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
# show the plot
plt.show()
```

Listing 25.1: Load the MNIST dataset in Keras

You can see that downloading and loading the MNIST dataset is as easy as calling the `mnist.load_data()` function. Running the above example, you should see the image below.

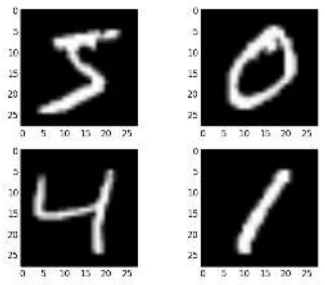


Figure 25.1: Examples from the MNIST dataset

25.3 Baseline Model with Multilayer Perceptrons

Do you really need a complex model like a convolutional neural network to get the best results with MNIST? You can get good results using a very simple neural network model with a single hidden layer. In this section, you will create a simple multilayer perceptron model that achieves an error rate of 1.74%. You will use this as a baseline for comparison to more complex convolutional neural network models. Let's start by importing the classes and functions you will need.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.utils import to_categorical
...
```

Listing 25.2: Import classes and functions

Now, you can load the MNIST dataset using the Keras helper function.

```
...
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Listing 25.3: Load the MNIST dataset

The training dataset is structured as a 3-dimensional array of instance, image width, and image height. For a multilayer perceptron model, you must reduce the images down into a vector of pixels. In this case, the 28×28-sized images will be 784 pixel input vectors. You can do this transform easily using the `reshape()` function on the NumPy array. The pixel values are integers, so they can be casted to floating point values and you can normalize them easily in the next step.

```
...
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape((X_train.shape[0], num_pixels)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], num_pixels)).astype('float32')
```

Listing 25.4: Prepare MNIST dataset for modeling

The pixel values are grayscale between 0 and 255. It is almost always a good idea to perform some scaling of input values when using neural network models. Because the scale is well known and well behaved, you can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

```
...
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
```

Listing 25.5: Normalize pixel values

Finally, the output variable is an integer from 0 to 9. This is a multiclass classification problem. As such, it is good practice to use a one-hot encoding of the class values, transforming the vector of class integers into a binary matrix. You can easily do this using the built-in `tf.keras.utils.to_categorical()` helper function in Keras.

```
...
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
```

Listing 25.6: One-hot encode the output variable

You are now ready to create your simple neural network model. You will define your model in a function. This is handy if you want to extend the example later and try and get a better score.

```
...
# define baseline model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_shape=(num_pixels,),
                    kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Listing 25.7: Baseline model for recognition

The model is a simple neural network with one hidden layer with the same number of neurons as there are inputs (784). A rectifier activation function is used for the neurons in the hidden layer. A softmax activation function is used on the output layer to turn the outputs into probability-like values and allow one class of the ten to be selected as the model's output prediction. Logarithmic loss is used as the loss function (called `categorical_crossentropy` in Keras), and the efficient ADAM gradient descent algorithm is used to learn the weights. A summary of the network structure is provided below:

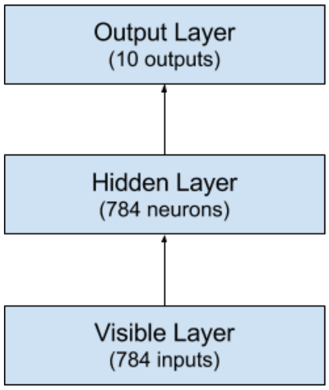


Figure 25.2: Summary of multilayer perceptron network structure

You can now fit and evaluate the model. The model is fit over ten epochs with updates every 200 images. The test data is used as the validation dataset, allowing you to see the skill of the model as it trains. A verbose value of 2 is used to reduce the output to one line for each training epoch. Finally, the test dataset is used to evaluate the model, and a classification error rate is printed.

```
...
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, epochs=10, batch_size=200,
          validation_data=(X_test, y_test), verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Listing 25.8: Evaluate the baseline model

After tying this all together, the complete code listing is provided below.

```
# Baseline MLP for MNIST dataset
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape((X_train.shape[0], num_pixels)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], num_pixels)).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
# define baseline model
```

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, input_shape=(num_pixels,),
                    kernel_initializer='normal', activation='relu'))
    model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, epochs=10, batch_size=200,
          validation_data=(X_test, y_test), verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Listing 25.9: Multilayer perceptron model for MNIST problem

Running the example might take a few minutes when you run it on a CPU. You should see the output below. This very simple network defined in very few lines of code achieves a respectable error rate of 2.3%.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

```
Epoch 1/10
300/300 - 1s - loss: 0.2792 - accuracy: 0.9215 - val_loss: 0.1387 - val_accuracy: 0.9590 - 1s/epoch
Epoch 2/10
300/300 - 1s - loss: 0.1113 - accuracy: 0.9676 - val_loss: 0.0923 - val_accuracy: 0.9709 - 929ms/epo
Epoch 3/10
300/300 - 1s - loss: 0.0704 - accuracy: 0.9799 - val_loss: 0.0728 - val_accuracy: 0.9787 - 912ms/epo
Epoch 4/10
300/300 - 1s - loss: 0.0502 - accuracy: 0.9859 - val_loss: 0.0664 - val_accuracy: 0.9808 - 904ms/epo
Epoch 5/10
300/300 - 1s - loss: 0.0356 - accuracy: 0.9897 - val_loss: 0.0636 - val_accuracy: 0.9803 - 905ms/epo
Epoch 6/10
300/300 - 1s - loss: 0.0261 - accuracy: 0.9932 - val_loss: 0.0591 - val_accuracy: 0.9813 - 907ms/epo
Epoch 7/10
300/300 - 1s - loss: 0.0195 - accuracy: 0.9953 - val_loss: 0.0564 - val_accuracy: 0.9828 - 910ms/epo
Epoch 8/10
300/300 - 1s - loss: 0.0145 - accuracy: 0.9969 - val_loss: 0.0580 - val_accuracy: 0.9810 - 954ms/epo
Epoch 9/10
300/300 - 1s - loss: 0.0116 - accuracy: 0.9973 - val_loss: 0.0594 - val_accuracy: 0.9817 - 947ms/epo
Epoch 10/10
300/300 - 1s - loss: 0.0079 - accuracy: 0.9985 - val_loss: 0.0735 - val_accuracy: 0.9770 - 914ms/epo
Baseline Error: 2.30%
```

Output 25.1: Sample output from evaluating the baseline model

25.4 Simple Convolutional Neural Network for MNIST

Now that you have seen how to load the MNIST dataset and train a simple multilayer perceptron model on it, it is time to develop a more sophisticated convolutional neural network or CNN model. Keras does provide a lot of capability for creating convolutional neural networks. In this section, you will create a simple CNN for MNIST that demonstrates how to use all the aspects of a modern CNN implementation, including convolutional layers, pooling layers, and dropout layers.

The first step is to import the classes and functions needed.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.utils import to_categorical
...
```

Listing 25.10: Import classes and functions

Next, you need to load the MNIST dataset and reshape it to be suitable for training a CNN. In Keras, the layers used for two-dimensional convolutions expect pixel values with the dimensions [*samples*][*width*][*height*][*channels*]. Note that you are forcing so-called channels-last ordering for consistency in this example. In the case of RGB, the last dimension channels would be 3 for the red, green, and blue components, and it would be like having three image inputs for every color image. In the case of MNIST, where the channels values are grayscale, the pixel dimension is set to 1.

```
...
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1).astype('float32')
```

Listing 25.11: Load dataset and separate into train and test sets

As before, it is a good idea to normalize the pixel values to the range 0 and 1 and one-hot encode the output variable.

```
...
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
```

Listing 25.12: Normalize and one-hot encode data

Next, define your neural network model. Convolutional neural networks are more complex than standard multilayer perceptrons, so you will start by using a simple structure that uses all the elements for state-of-the-art results. Below summarizes the network architecture.

1. The first hidden layer is a convolutional layer called a `Conv2D`. The layer has 32 feature maps, with the size of 5×5 and a rectifier activation function. This is the input layer that expects images with the structure outline above.
2. Next, we define a pooling layer that takes the max called `MaxPooling2D`. It is configured with a pool size of 2×2.
3. The next layer is a regularization layer using dropout called `Dropout`. It is configured to randomly exclude 20% of neurons in the layer in order to reduce overfitting.
4. Next is a layer that converts the 2D matrix data to a vector called `Flatten`. It allows the output to be processed by standard, fully connected layers.
5. Next is a fully connected layer with 128 neurons and a rectifier activation function is used.
6. Finally, the output layer has ten neurons for the ten classes and a softmax activation function to output probability-like predictions for each class.

As before, the model is trained using logarithmic loss and the ADAM gradient descent algorithm. A depiction of the network structure is provided below.

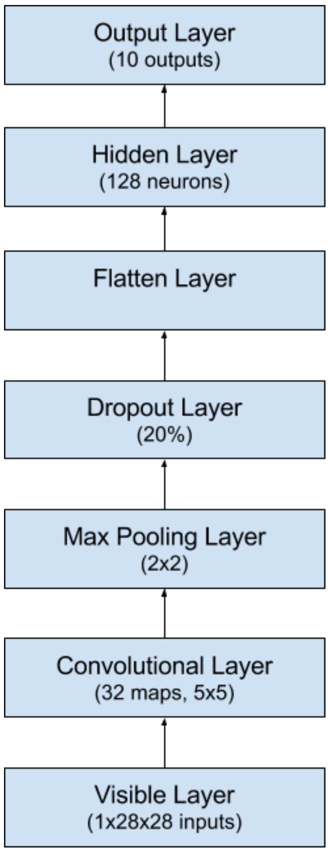


Figure 25.3: Summary of convolutional neural network structure

```
...
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(28, 28, 1), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Listing 25.13: Define and compile CNN model

You evaluate the model the same way as before with the multilayer perceptron. The CNN is fit over ten epochs with a batch size of 200.

```
...
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 25.14: Fit and evaluate the CNN model

After tying this all together, the complete example is listed below.

```
# Simple CNN for the MNIST Dataset
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.utils import to_categorical
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1)).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
# define a simple CNN model
```

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(28, 28, 1), activation='relu'))
    model.add(MaxPooling2D())
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 25.15: CNN model for MNIST problem

After running the example, the accuracy of the training and validation test is printed for each epoch, and at the end, the classification error rate is printed.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Epochs may take about 45 seconds to run on the GPU (e.g., on AWS). You can see that the network achieves an error rate of 1.19%, which is better than our simple multilayer perceptron model above.

```
Epoch 1/10
300/300 [==============================] - 4s 12ms/step - loss: 0.2372 - accuracy: 0.9344 - val_loss
Epoch 2/10
300/300 [==============================] - 4s 13ms/step - loss: 0.0697 - accuracy: 0.9786 - val_loss
Epoch 3/10
300/300 [==============================] - 4s 13ms/step - loss: 0.0483 - accuracy: 0.9854 - val_loss
Epoch 4/10
300/300 [==============================] - 4s 13ms/step - loss: 0.0366 - accuracy: 0.9887 - val_loss
Epoch 5/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0300 - accuracy: 0.9909 - val_loss
Epoch 6/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0241 - accuracy: 0.9927 - val_loss
Epoch 7/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0210 - accuracy: 0.9932 - val_loss
Epoch 8/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0167 - accuracy: 0.9945 - val_loss
Epoch 9/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0142 - accuracy: 0.9956 - val_loss
Epoch 10/10
```

```
300/300 [==============================] - 4s 14ms/step - loss: 0.0114 - accuracy: 0.9966 - val_loss
CNN Error: 1.19%
```

Output 25.2: Sample output from evaluating the CNN model

25.5 Larger Convolutional Neural Network for MNIST

Now that you have seen how to create a simple CNN, let's take a look at a model capable of close to state-of-the-art results. You will import the classes and functions, then load and prepare the data the same as in the previous CNN example.

```
# Larger CNN for the MNIST Dataset
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.utils import to_categorical
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1)).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
...
```

Listing 25.16: Import classes and functions for the larger CNN model

This time you will define a large CNN architecture with additional convolutional, max pooling layers, and fully connected layers. The network topology can be summarized as follows:

1. Convolutional layer with 30 feature maps of size 5×5
2. Pooling layer taking the max over 2×2 patches
3. Convolutional layer with 15 feature maps of size 3×3
4. Pooling layer taking the max over 2×2 patches
5. Dropout layer with a probability of 20%
6. Flatten layer
7. Fully connected layer with 128 neurons and rectifier activation
8. Fully connected layer with 50 neurons and rectifier activation
9. Output layer

A depictions of this larger network structure is provided below.

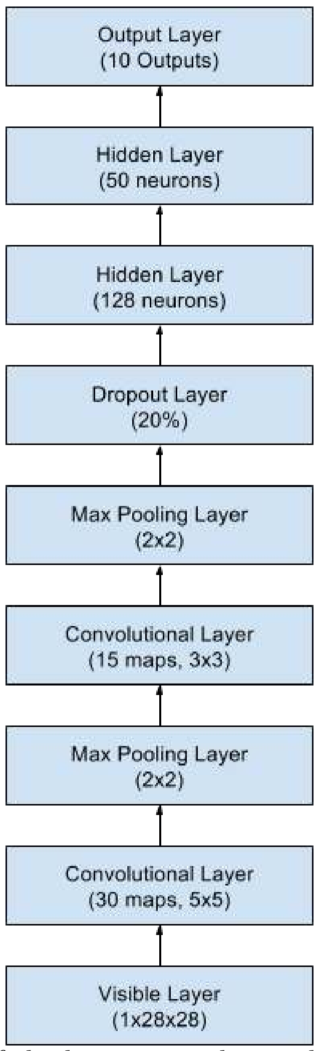


Figure 25.4: Summary of the larger convolutional neural network structure

```
...
# define the larger model
def larger_model():
    # create model
    model = Sequential()
    model.add(Conv2D(30, (5, 5), input_shape=(28, 28, 1), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(15, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Listing 25.17: Defining the larger convolutional neural network

Like the previous two experiments, the model is fit over ten epochs with a batch size of 200.

```
...
# build the model
model = larger_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 25.18: Fitting the larger CNN model

After tying this all together, the complete example is listed below.

```
# Larger CNN for the MNIST Dataset
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.utils import to_categorical
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1)).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
# define the larger model
def larger_model():
    # create model
    model = Sequential()
    model.add(Conv2D(30, (5, 5), input_shape=(28, 28, 1), activation='relu'))
    model.add(MaxPooling2D())
    model.add(Conv2D(15, (3, 3), activation='relu'))
    model.add(MaxPooling2D())
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
# build the model
model = larger_model()
```

```
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, batch_size=200)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 25.19: Larger CNN for the MNIST problem

Running the example prints accuracy on the training and validation datasets of each epoch and a final classification error rate.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

The model takes about 100 seconds to run per epoch. This slightly larger model achieves a respectable classification error rate of 0.83%.

```
Epoch 1/10
300/300 [==============================] - 4s 14ms/step - loss: 0.4104 - accuracy: 0.8727 - val_loss
Epoch 2/10
300/300 [==============================] - 5s 15ms/step - loss: 0.1062 - accuracy: 0.9669 - val_loss
Epoch 3/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0771 - accuracy: 0.9765 - val_loss
Epoch 4/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0624 - accuracy: 0.9812 - val_loss
Epoch 5/10
300/300 [==============================] - 4s 15ms/step - loss: 0.0521 - accuracy: 0.9838 - val_loss
Epoch 6/10
300/300 [==============================] - 4s 15ms/step - loss: 0.0453 - accuracy: 0.9861 - val_loss
Epoch 7/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0415 - accuracy: 0.9866 - val_loss
Epoch 8/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0376 - accuracy: 0.9879 - val_loss
Epoch 9/10
300/300 [==============================] - 4s 14ms/step - loss: 0.0327 - accuracy: 0.9895 - val_loss
Epoch 10/10
300/300 [==============================] - 4s 15ms/step - loss: 0.0294 - accuracy: 0.9904 - val_loss
Large CNN Error: 0.90%
```

Output 25.3: Sample output from evaluating the larger CNN model

This is not an optimized network topology. Nor is it a reproduction of a network topology from a recent paper. There is a lot of opportunity for you to tune and improve upon this model. What is the best error rate score you can achieve?

25.6 Resources on MNIST

The MNIST dataset is very well studied. Below are some additional resources you might want to look into.

Articles

Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. *The MNIST database of handwritten digits.*
http://yann.lecun.com/exdb/mnist/

Rodrigo Benenson. *What is the class of this image?* Classification datasets results, 2016.
https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

Digit Recognizer: Learn computer vision fundamentals with the famous MNIST data. Kaggle.
https://www.kaggle.com/c/digit-recognizer

Hubert Eichner. *Neural Net for Handwritten Digit Recognition in JavaScript.*
http://myselph.de/neuralNet.html

25.7 Summary

In this chapter, you discovered the MNIST handwritten digit recognition problem and deep learning models developed in Python using the Keras library that are capable of achieving excellent results. Working through this chapter, you learned:

- ▷ How to load the MNIST dataset in Keras and generate plots of the dataset
- ▷ How to reshape the MNIST dataset and develop a simple but well-performing multilayer perceptron model for the problem
- ▷ How to use Keras to create convolutional neural network models for MNIST
- ▷ How to develop and evaluate larger CNN models for MNIST capable of near world-class results

In the next chapter, you will learn some image preprocessing techniques to improve the model accuracy.

26 Improve Model Accuracy with Image Augmentation

Data preparation is required when working with neural networks and deep learning models. Increasingly, data augmentation is also required on more complex object recognition tasks. This is because we can increase the variation of the input images so the network can be smarter from learning with more data. In this chapter, you will discover how to use data preparation and data augmentation with your image datasets when developing and evaluating deep learning models in Python with Keras. After reading this chapter, you will know:

- ▷ About the image augmentation API provided by Keras and how to use it with your models
- ▷ How to perform feature standardization
- ▷ How to perform ZCA whitening of your images
- ▷ How to augment data with random rotations, shifts and flips of images
- ▷ How to save augmented image data to disk

Let's get started.

Overview

This chapter is divided into nine sections; they are:

- ▷ Keras Image Augmentation API
- ▷ Point of Comparison for Image Augmentation
- ▷ Feature Standardization
- ▷ ZCA Whitening
- ▷ Random Rotations
- ▷ Random Shifts
- ▷ Random Flips
- ▷ Saving Augmented Images to File
- ▷ Tips For Augmenting Image Data with Keras

26.1 Keras Image Augmentation API

Like the rest of Keras, the image augmentation API is simple and powerful. Keras provides the `ImageDataGenerator` class that defines the configuration for image data preparation and augmentation. This includes capabilities such as:

- ▷ Sample-wise standardization
- ▷ Feature-wise standardization
- ▷ ZCA whitening
- ▷ Random rotation, shifts, shear and flips
- ▷ Dimension reordering
- ▷ Save augmented images to disk

An augmented image generator can be created as follows:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator()
```

Listing 26.1: Create a `ImageDataGenerator`

Rather than performing the operations on your entire image dataset in memory, the API is designed to be iterated by the deep learning model fitting process, creating augmented image data for you just in time. This reduces your memory overhead but adds some additional time cost during model training. After you have created and configured your `ImageDataGenerator`, you must fit it on your data. This will calculate any statistics required to actually perform the transforms to your image data. You can do this by calling the `fit()` function on the data generator and passing it to your training dataset.

```
datagen.fit(train)
```

Listing 26.2: Fit the `ImageDataGenerator`

The data generator itself is, in fact, an iterator, returning batches of image samples when requested. You can configure the batch size and prepare the data generator and get batches of images by calling the `flow()` function.

```
X_batch, y_batch = datagen.flow(train, train, batch_size=32)
```

Listing 26.3: Configure the batch size for the `ImageDataGenerator`

Finally, you can make use of the data generator. Instead of calling the `fit()` function on your model, you must call the `fit_generator()` function and pass in the data generator and the desired length of an epoch as well as the total number of epochs on which to train.

```
fit_generator(datagen, samples_per_epoch=len(train), epochs=100)
```

Listing 26.4: Fit a model using `ImageDataGenerator`

You can learn more about the Keras image data generator API in the Keras documentation.

26.2 Point of Comparison for Image Augmentation

Now that you know how the image augmentation API in Keras works, let's look at some examples. We will use the MNIST handwritten digit recognition task in these examples (see Section 25.1). To begin, let's take a look at the first nine images in the training dataset.

```
# Plot images
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
# load dbata
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# create a grid of 3x3 images
fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(4,4))
for i in range(3):
    for j in range(3):
        ax[i][j].imshow(X_train[i*3+j], cmap=plt.get_cmap("gray"))
# show the plot
plt.show()
```

Listing 26.5: Load and plot the MNIST dataset

Running this example provides the following image that you can use as a point of comparison with the image preparation and augmentation tasks in the examples below.

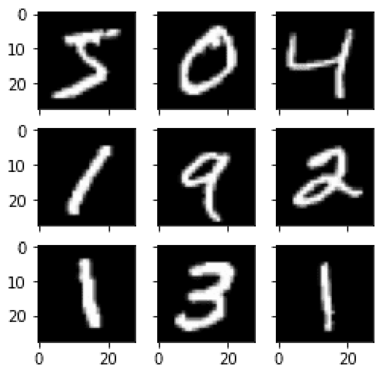


Figure 26.1: Samples from the MNIST dataset.

26.3 Feature Standardization

It is also possible to standardize pixel values across the entire dataset. This is called feature standardization and mirrors the type of standardization often performed for each column in a tabular dataset.

You can perform feature standardization by setting the `featurewise_center` and `featurewise_std_normalization` arguments to True on the `ImageDataGenerator` class. These are set to False by default. However, the recent version of Keras has a bug in the feature standardization so that the mean and standard deviation is calculated across all pixels. If you use the `fit()` function from the `ImageDataGenerator` class, you will see an image similar to the one above:

```
# Standardize images across the dataset, mean=0, stdev=1
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(featurewise_center=True, featurewise_std_normalization=True)
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9, shuffle=False):
    print(X_batch.min(), X_batch.mean(), X_batch.max())
    # create a grid of 3x3 images
    fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(4,4))
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(X_batch[i*3+j].reshape(28,28), cmap=plt.get_cmap("gray"))
    # show the plot
    plt.show()
    break
```

Listing 26.6: Standardize images across the dataset

For example, the minimum, mean, and maximum values from the batch printed above are:

```
-0.42407447 -0.04093817 2.8215446
```

Output 26.1: Minimum, mean, and maximum pixel value after standardization in Listing 26.6

And the image displayed is as follows:

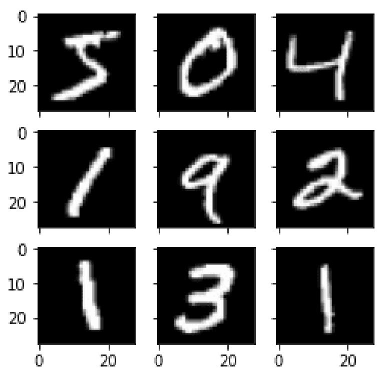


Figure 26.2: Image from feature-wise standardization

The workaround is to compute the feature standardization manually. Each pixel should have a separate mean and standard deviation, and it should be computed across different samples but independent from other pixels in the same sample. You just need to replace the `fit()` function with your own computation:

```
# Standardize images across the dataset, every pixel has mean=0, stdev=1
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(featurewise_center=True, featurewise_std_normalization=True)
# fit parameters from data
datagen.mean = X_train.mean(axis=0)
datagen.std = X_train.std(axis=0)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9, shuffle=False):
    print(X_batch.min(), X_batch.mean(), X_batch.max())
    # create a grid of 3x3 images
    fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(4,4))
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(X_batch[i*3+j].reshape(28,28), cmap=plt.get_cmap("gray"))
    # show the plot
    plt.show()
    break
```

Listing 26.7: Feature-wise standardizationn

The minimum, mean, and maximum as printed now have a wider range:

```
-1.2742625 -0.028436039 17.46127
```

Output 26.2: Minimum, mean. maximum pixel value after standardization in Listing 26.7

Running this example, you can see that the effect is different, seemingly darkening and lightening different digits.

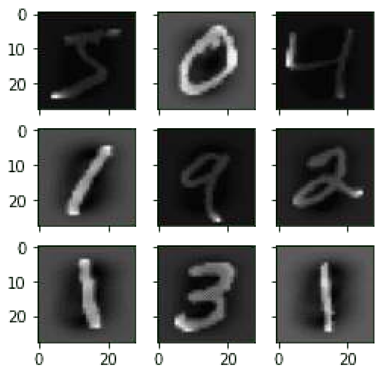


Figure 26.3: MNIST images with standardized features

26.4 ZCA Whitening

A whitening transform of an image is a linear algebraic operation that reduces the redundancy in the matrix of pixel images. Less redundancy in the image is intended to better highlight the structures and features in the image to the learning algorithm. Typically, image whitening is performed using the Principal Component Analysis (PCA) technique. More recently, an alternative called ZCA (learn more in Appendix A of Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*) shows better results in transformed images that keep all the original dimensions. And unlike PCA, the resulting transformed images still look like their originals. Precisely, whitening converts each image into a white noise vector, i.e., each element in the vector has zero mean unit standard derivation and is statistically independent of each other. You can perform a ZCA whitening transform by setting the `zca_whitening` argument to `True`. But due to the same issue as feature standardization, you must first zero-center your input data separately:

```
# ZCA Whitening
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(featurewise_center=True,
                             featurewise_std_normalization=True,
                             zca_whitening=True)
# fit parameters from data
X_mean = X_train.mean(axis=0)
datagen.fit(X_train - X_mean)
# configure batch size and retrieve one batch of images
```

```
X_centered = X_train - X_mean
for X_batch, y_batch in datagen.flow(X_centered, y_train, batch_size=9, shuffle=False):
    print(X_batch.min(), X_batch.mean(), X_batch.max())
    # create a grid of 3x3 images
    fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(4,4))
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(X_batch[i*3+j].reshape(28,28), cmap=plt.get_cmap("gray"))
    # show the plot
    plt.show()
    break
```

Listing 26.8: Example of ZCA whitening

Running the example, you can see the same general structure in the images and how the outline of each digit has been highlighted.

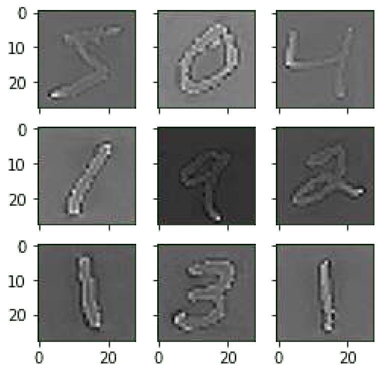


Figure 26.4: ZCA whitening MNIST images

26.5 Random Rotations

Sometimes images in your sample data may have varying and different rotations in the scene. You can train your model to better handle rotations of images by artificially and randomly rotating images from your dataset during training. The example below creates random rotations of the MNIST digits up to 90 degrees by setting the `rotation_range` argument.

```
# Random Rotations
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
```

```
datagen = ImageDataGenerator(rotation_range=90)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9, shuffle=False):
    # create a grid of 3x3 images
    fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(4,4))
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(X_batch[i*3+j].reshape(28,28), cmap=plt.get_cmap("gray"))
    # show the plot
    plt.show()
    break
```

Listing 26.9: Example of random image rotations

Running the example, you can see that images have been rotated left and right up to a limit of 90 degrees. This is not helpful on this problem because the MNIST digits have a normalized orientation, but this transform might be of help when learning from photographs where the objects may have different orientations.

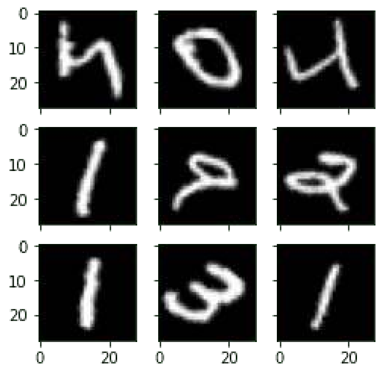


Figure 26.5: Random rotations of MNIST images

26.6 Random Shifts

Objects in your images may not be centered in the frame. They may be off-center in a variety of different ways. You can train your deep learning network to expect and currently handle off-center objects by artificially creating shifted versions of your training data. Keras supports separate horizontal and vertical random shifting of training data by the `width_shift_range` and `height_shift_range` arguments.

```
# Random Shifts
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
```

```
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
shift = 0.2
datagen = ImageDataGenerator(width_shift_range=shift, height_shift_range=shift)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9, shuffle=False):
    # create a grid of 3x3 images
    fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(4,4))
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(X_batch[i*3+j].reshape(28,28), cmap=plt.get_cmap("gray"))
    # show the plot
    plt.show()
    break
```

Listing 26.10: Example of random image shifts

Running this example creates shifted versions of the digits. Again, this is not required for MNIST as the handwritten digits are already centered, but you can see how this might be useful on more complex problem domains.

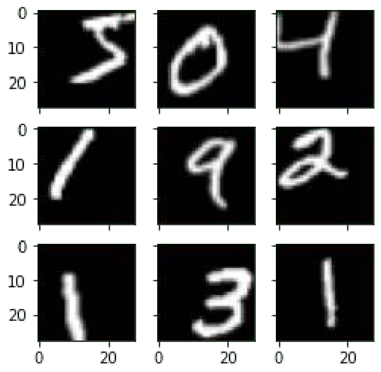


Figure 26.6: Random shifted MNIST images

26.7 Random Flips

Another augmentation to your image data that can improve performance on large and complex problems is to create random flips of images in your training data. Keras supports random flipping along both the vertical and horizontal axes using the `vertical_flip` and `horizontal_flip` arguments.

```
# Random Flips
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9, shuffle=False):
    # create a grid of 3x3 images
    fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(4,4))
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(X_batch[i*3+j].reshape(28,28), cmap=plt.get_cmap("gray"))
    # show the plot
    plt.show()
    break
```

Listing 26.11: Example of random image flips

Running this example, you can see flipped digits. Flipping digits in MNIST is not useful as they will always have the correct left and right orientation, but this may be useful for problems with photographs of objects in a scene that can have a varied orientation.

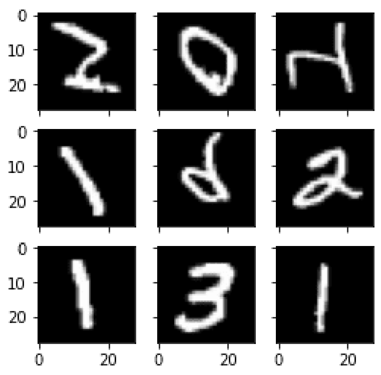


Figure 26.7: Randomly flipped MNIST images

26.8 Saving Augmented Images to File

The data preparation and augmentation are performed just-in-time by Keras. This is efficient in terms of memory, but you may require the exact images used during training. For example, perhaps you would like to use them with a different software package later or only generate them once and use them on multiple different deep learning models or configurations.

Keras allows you to save the images generated during training. The directory, filename prefixes, and image file type can be specified to the `flow()` function before training. Then, during training, the generated images will be written to the file. The example below demonstrates this and writes nine images to a `images` subdirectory (you need to create this directory first) with the prefix `aug` and the file type of PNG.

```python
# Save augmented images to file
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9, shuffle=False,
                                     save_to_dir='images', save_prefix='aug', save_format='png'):
    # create a grid of 3x3 images
    fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(4,4))
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(X_batch[i*3+j].reshape(28,28), cmap=plt.get_cmap("gray"))
    # show the plot
    plt.show()
    break
```

Listing 26.12: Save augmented images to file

Running the example, you can see that images are only written when they are generated.

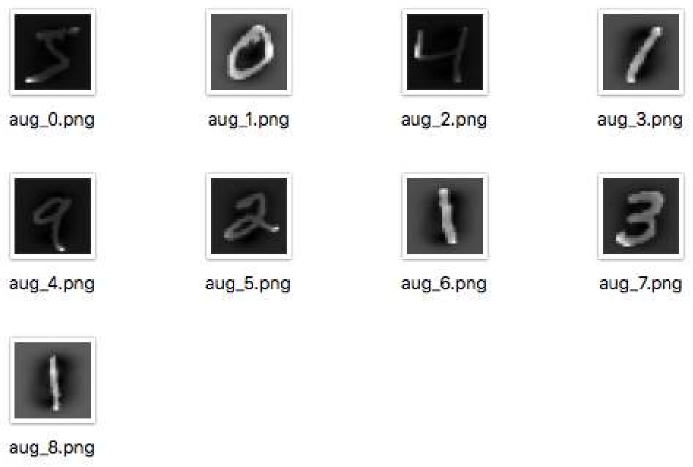


Figure 26.8: Augmented MNIST images saved to file

26.9 Tips for Augmenting Image Data with Keras

Image data is unique in that you can review the data and transformed copies of the data and quickly get an idea of how the inputs may perceived it.

Below are some tips for getting the most from image data preparation and augmentation for deep learning.

- ▷ **Review Dataset**. Take some time to review your dataset in great detail. Look at the images. Take note of image preparation and augmentations that might benefit the training process of your model, such as the need to handle different shifts, rotations, or flips of objects in the scene.
- ▷ **Review Augmentations**. Review sample images after the augmentation has been performed. It is one thing to intellectually know what image transforms you are using; it is a very different thing to look at examples. Review images both with individual augmentations you are using as well as the full set of augmentations you plan to use in aggregate. You may see ways to simplify or further enhance your model training process.
- ▷ **Evaluate a Suite of Transforms**. Try more than one image data preparation and augmentation scheme. Often you can be surprised by the results of a data preparation scheme you did not think would be beneficial.

26.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

APIs

tf.keras.preprocessing.image.ImageDataGenerator. Tensorflow API.
https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

Image data preprocessing. Keras API reference.
https://keras.io/api/preprocessing/image/

Articles

Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech Report TR-2009. University of Toronto, Apr. 2009.
https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

Whitening transformation. Wikipedia.
https://en.wikipedia.org/wiki/Whitening_transformation

PCA Whitening. Unsupervised Feature Learning and Deep Learning Tutorial.
http://ufldl.stanford.edu/tutorial/unsupervised/PCAWhitening/

Alex Krizhevsky. *The CIFAR-10 dataset*.
https://www.cs.toronto.edu/~kriz/cifar.html

26.11 Summary

In this chapter, you discovered image data preparation and augmentation.

You discovered a range of techniques you can use easily in Python with Keras for deep learning models. You learned about:

- The `ImageDataGenerator` API in Keras for generating transformed images just-in-time
- Sample-wise and feature-wise pixel standardization
- The ZCA whitening transform
- Random rotations, shifts and flips of images
- How to save transformed images to file for later reuse

In the next chapter, you will see more ways to do image augmentation.

Image Augmentation with Keras Preprocessing Layers and tf.image

27

When you work on a machine learning problem related to images, not only do you need to collect some images as training data, but you also need to employ augmentation to create variations in the image. It is especially true for more complex object recognition problems.

There are many ways for image augmentation. You may use some external libraries or write your own functions for that. There are some modules in TensorFlow and Keras for augmentation too. In this chapter, you will discover how you can use the Keras preprocessing layer as well as the `tf.image` module in TensorFlow for image augmentation. After reading this chapter, you will know:

- ▷ What are the Keras preprocessing layers, and how to use them
- ▷ What are the functions provided by the `tf.image` module for image augmentation
- ▷ How to use augmentation together with the `tf.data` dataset

Let's get started.

Overview

This chapter is divided into five sections; they are:

- ▷ Getting Images
- ▷ Visualizing the Images
- ▷ Keras Preprocessing Layers
- ▷ Using tf.image API for Augmentation
- ▷ Using Preprocessing Layers in Neural Networks

27.1 Getting Images

Before you see how you can do augmentation, you need to get the images. Ultimately, you need the images to be represented as arrays, for example, in $H \times W \times 3$ in 8-bit integers for the RGB pixel value. There are many ways to get the images. Some can be downloaded as a ZIP file.

If you're using TensorFlow, you may get some image datasets from the `tensorflow_datasets` library.

In this chapter, you will use the citrus leaves images, which is a small dataset of less than 100MB. It can be downloaded from `tensorflow_datasets` as follows:

```
import tensorflow_datasets as tfds
ds, meta = tfds.load('citrus_leaves', with_info=True, split='train', shuffle_files=True)
```

Listing 27.1: Loading the citrus leaves dataset

Running this code the first time will download the image dataset into your computer with the following output:

```
Downloading and preparing dataset 63.87 MiB (download: 63.87 MiB, generated: 37.89 MiB, total:
↪  101.76 MiB) to ~/tensorflow_datasets/citrus_leaves/0.1.2...
Extraction completed...: 100%|██████████████████████████████| 1/1 [00:06<00:00,  6.54s/ file]
Dl Size...: 100%|██████████████████████████████████████████| 63/63 [00:06<00:00,  9.63 MiB/s]
Dl Completed...: 100%|█████████████████████████████████████| 1/1 [00:06<00:00,  6.54s/ url]
Dataset citrus_leaves downloaded and prepared to ~/tensorflow_datasets/citrus_leaves/0.1.2.
↪  Subsequent calls will reuse this data.
```

Output 27.1: Downloading the citrus leaves dataset the first time

The function above returns the images as a `tf.data` dataset object and the metadata. This is a classification dataset. You can print the training labels with the following:

```
...
for i in range(meta.features['label'].num_classes):
    print(meta.features['label'].int2str(i))
```

Listing 27.2: Print the classification labels of the citrus leaves dataset

This prints:

```
Black spot
canker
greening
healthy
```

Output 27.2: Classification labels of the citrus leaves dataset

If you run this code again at a later time, you will reuse the downloaded image. But the other way to load the downloaded images into a `tf.data` dataset is to use the `image_dataset_from_directory()` function.

As you can see from the screen output above, the dataset is downloaded into the directory `~/tensorflow_datasets`. If you look at the directory, you see the directory structure as follows:

```
.../Citrus/Leaves
├── Black spot
├── Melanose
├── canker
├── greening
└── healthy
```

Output 27.3: Directory structure of the downloaded dataset

The directories are the labels, and the images are files stored under their corresponding directory. You can let the function to read the directory recursively into a dataset:

```python
import tensorflow as tf
from tensorflow.keras.utils import image_dataset_from_directory

# set to fixed image size 256x256
PATH = ".../Citrus/Leaves"
ds = image_dataset_from_directory(PATH,
                                  validation_split=0.2, subset="training",
                                  image_size=(256,256), interpolation="bilinear",
                                  crop_to_aspect_ratio=True,
                                  seed=42, shuffle=True, batch_size=32)
```

Listing 27.3: Load the citrus leaves dataset from local hard disk

You may want to set `batch_size=None` if you do not want the dataset to be batched. Usually, you want the dataset to be batched for training a neural network model.

27.2 Visualizing the Images

It is important to visualize the augmentation result, so you can verify the augmentation result is what we want it to be. You can use Matplotlib for this. In Matplotlib, you have the `imshow()` function to display an image. However, for the image to be displayed correctly, the image should be presented as an array of 8-bit unsigned integers (uint8). Given that you have a dataset created using `image_dataset_from_directory()`. You can get the first batch (of 32 images) and display a few of them using `imshow()`, as follows:

```python
...
import matplotlib.pyplot as plt

fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(5,5))

for images, labels in ds.take(1):
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(images[i*3+j].numpy().astype("uint8"))
            ax[i][j].set_title(ds.class_names[labels[i*3+j]])
plt.show()
```

Listing 27.4: Show example images from a dataset

Here, you see a display of nine images in a grid, labeled with their corresponding classification label, using `ds.class_names`. The images should be converted to NumPy array in uint8 for display. This code displays an image like the following:

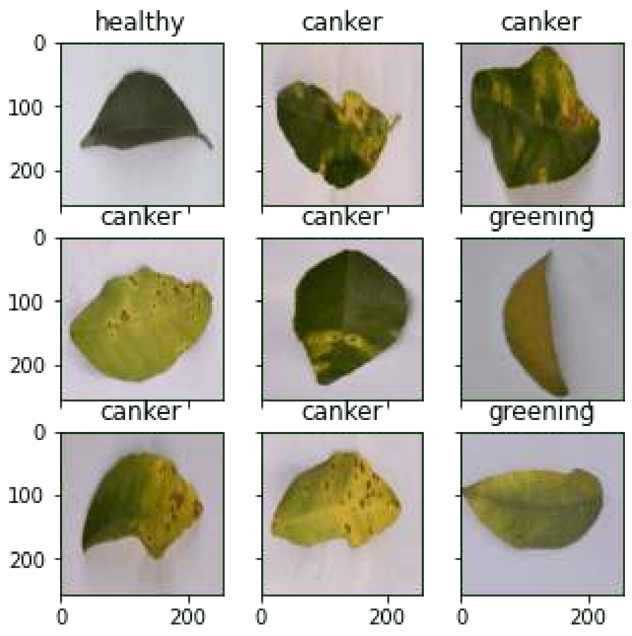


Figure 27.1: Example images from the citrus leaves dataset

The complete code from loading the image to display is as follows:

```
from tensorflow.keras.utils import image_dataset_from_directory
import matplotlib.pyplot as plt

# use image_dataset_from_directory() to load images, with image size scaled to 256x256
PATH='.../Citrus/Leaves'  # modify to your path
ds = image_dataset_from_directory(PATH,
                                  validation_split=0.2, subset="training",
                                  image_size=(256,256), interpolation="mitchellcubic",
                                  crop_to_aspect_ratio=True,
                                  seed=42, shuffle=True, batch_size=32)

# Take one batch from dataset and display the images
fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(5,5))

for images, labels in ds.take(1):
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(images[i*3+j].numpy().astype("uint8"))
            ax[i][j].set_title(ds.class_names[labels[i*3+j]])
plt.show()
```

Listing 27.5: Loading images from disk and display a few samples

Note that if you're using `tensorflow_datasets` to get the image, the samples are presented as a dictionary instead of a tuple of (image,label). You should change your code slightly to the following:

```
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt

# use tfds.load() or image_dataset_from_directory() to load images
ds, meta = tfds.load('citrus_leaves', with_info=True, split='train', shuffle_files=True)
ds = ds.batch(32)

# Take one batch from dataset and display the images
fig, ax = plt.subplots(3, 3, sharex=True, sharey=True, figsize=(5,5))

for sample in ds.take(1):
    images, labels = sample["image"], sample["label"]
    for i in range(3):
        for j in range(3):
            ax[i][j].imshow(images[i*3+j].numpy().astype("uint8"))
            ax[i][j].set_title(meta.features['label'].int2str(labels[i*3+j]))
plt.show()
```

Listing 27.6: Loading images from `tensorflow_datasets` *and display a few samples*

Note: For the rest of this chapter, assume the dataset is created using `image_dataset_from_directory()`. You may need to tweak the code slightly if your dataset is created differently.

27.3 Keras Preprocessing Layers

Keras comes with many neural network layers, such as convolution layers, that you need to train. There are also layers with no parameters to train, such as flatten layers to convert an array like an image into a vector.

The preprocessing layers in Keras are specifically designed to use in the early stages of a neural network. You can use them for image preprocessing, such as to resize or rotate the image or adjust the brightness and contrast. While the preprocessing layers are supposed to be part of a larger neural network, you can also use them as functions. Below is how you can use the resizing layer as a function to transform some images and display them side-by-side with the original:

```
...

# create a resizing layer
out_height, out_width = 128,256
resize = tf.keras.layers.Resizing(out_height, out_width)

# show original vs resized
fig, ax = plt.subplots(2, 3, figsize=(6,4))

for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # resize
        ax[1][i].imshow(resize(images[i]).numpy().astype("uint8"))
        ax[1][i].set_title("resize")
plt.show()
```

Listing 27.7: Resize images

The images are in 256×256 pixels, and the resizing layer will make them into 256×128 pixels. The output of the above code is as follows:

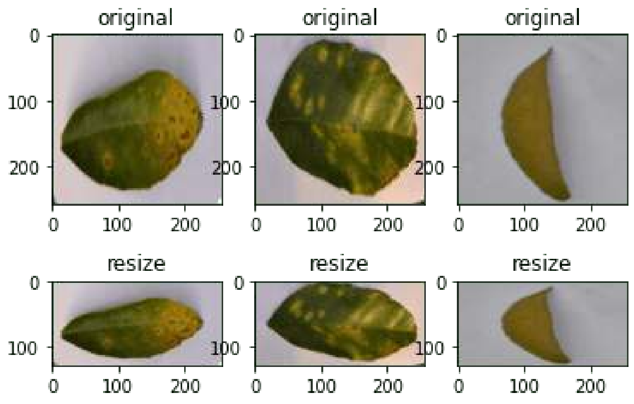


Figure 27.2: Example of images and their resize transform

Since the resizing layer is a function, you can chain them to the dataset itself. For example,

```
...
def augment(image, label):
    return resize(image), label

resized_ds = ds.map(augment)

for image, label in resized_ds:
   ...
```

Listing 27.8: Attach resize layer into dataset

The dataset `ds` has samples in the form of `(image, label)`. Hence you created a function that takes in such tuple and preprocesses the image with the resizing layer. You then assigned this function as an argument for the `map()` in the dataset. When you draw a sample from the new dataset created with the `map()` function, the image will be a transformed one.

There are more preprocessing layers available. Some are demonstrated below.

The examples below are tested with TensorFlow 2.9. Some of the preprocessing functions may not be availble in earlier version of TensorFlow.

As we saw above, we can resize the image. We can also randomly enlarge or shrink the height or width of an image. Similarly, we can zoom in or zoom out on an image. Below is an example to manipulate the image size in various ways for a maximum of 30% increase or decrease:

```
...

# Create preprocessing layers
out_height, out_width = 128,256
resize = tf.keras.layers.Resizing(out_height, out_width)
height = tf.keras.layers.RandomHeight(0.3)
width = tf.keras.layers.RandomWidth(0.3)
zoom = tf.keras.layers.RandomZoom(0.3)

# Visualize images and augmentations
fig, ax = plt.subplots(5, 3, figsize=(6,14))

for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # resize
        ax[1][i].imshow(resize(images[i]).numpy().astype("uint8"))
        ax[1][i].set_title("resize")
        # height
        ax[2][i].imshow(height(images[i]).numpy().astype("uint8"))
        ax[2][i].set_title("height")
        # width
        ax[3][i].imshow(width(images[i]).numpy().astype("uint8"))
        ax[3][i].set_title("width")
        # zoom
        ax[4][i].imshow(zoom(images[i]).numpy().astype("uint8"))
        ax[4][i].set_title("zoom")
plt.show()
```

Listing 27.9: Transform images by resize, adjusting height, adjusting width, and zoom

This code shows images as in Figure 27.3. While you specified a fixed dimension in resize, you have a random amount of manipulation in other augmentations.

You can also do flipping, rotation, cropping, and geometric translation using preprocessing layers:

```
...
# Create preprocessing layers
flip = tf.keras.layers.RandomFlip("horizontal_and_vertical") # or "horizontal", "vertical"
rotate = tf.keras.layers.RandomRotation(0.2)
crop = tf.keras.layers.RandomCrop(out_height, out_width)
translation = tf.keras.layers.RandomTranslation(height_factor=0.2, width_factor=0.2)

# Visualize augmentations
fig, ax = plt.subplots(5, 3, figsize=(6,14))

for images, labels in ds.take(1):
    for i in range(3):
```

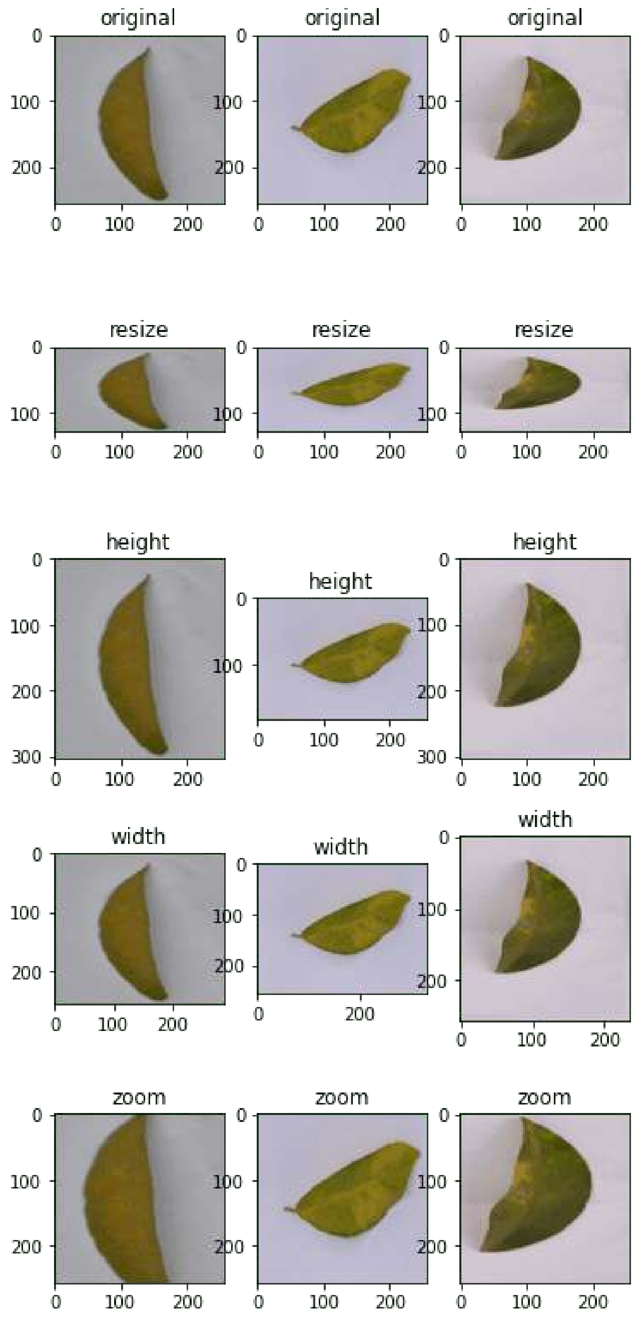


Figure 27.3: Example transformation from Listing 27.9

```
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # flip
        ax[1][i].imshow(flip(images[i]).numpy().astype("uint8"))
        ax[1][i].set_title("flip")
        # crop
        ax[2][i].imshow(crop(images[i]).numpy().astype("uint8"))
        ax[2][i].set_title("crop")
        # translation
        ax[3][i].imshow(translation(images[i]).numpy().astype("uint8"))
```

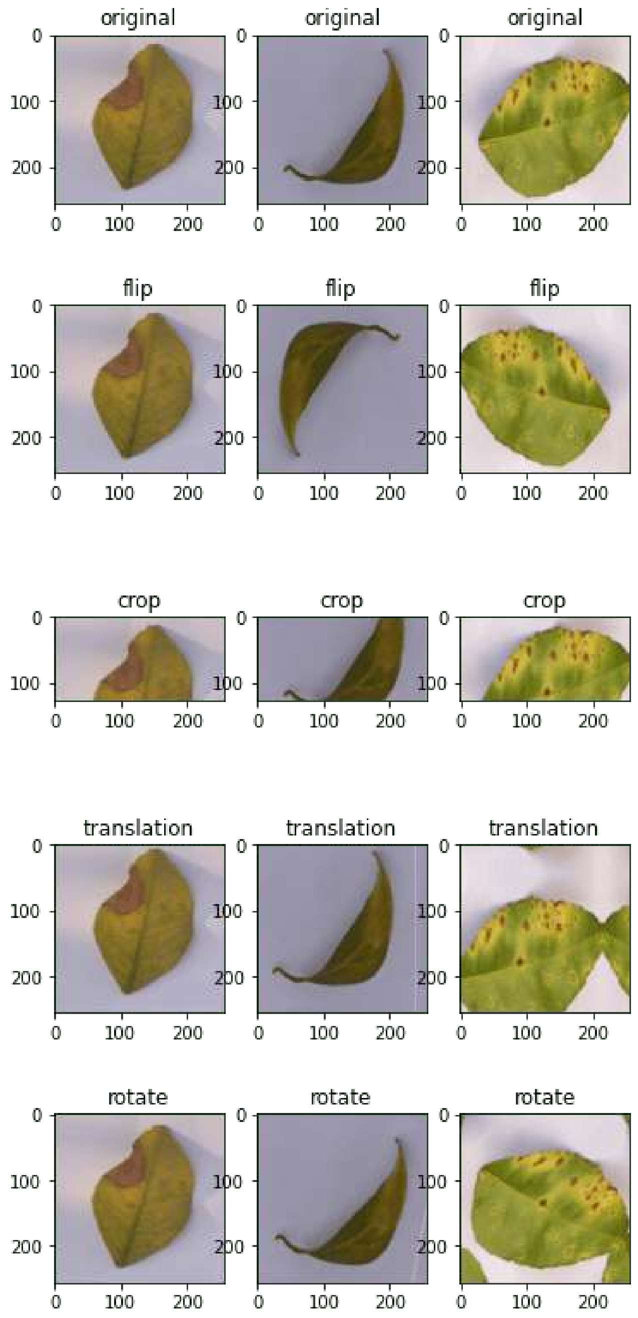


Figure 27.4: Example transformation from Listing 27.10

```
        ax[3][i].set_title("translation")
        # rotate
        ax[4][i].imshow(rotate(images[i]).numpy().astype("uint8"))
        ax[4][i].set_title("rotate")
plt.show()
```

Listing 27.10: Transform images by flipping, cropping, translation, and rotation

This code shows the images as in Figure 27.4.

And finally, you can do augmentations on color adjustments as well:

```
...
brightness = tf.keras.layers.RandomBrightness([-0.8,0.8])
contrast = tf.keras.layers.RandomContrast(0.2)

# Visualize augmentation
fig, ax = plt.subplots(3, 3, figsize=(6,7))

for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # brightness
        ax[1][i].imshow(brightness(images[i]).numpy().astype("uint8"))
        ax[1][i].set_title("brightness")
        # contrast
        ax[2][i].imshow(contrast(images[i]).numpy().astype("uint8"))
        ax[2][i].set_title("contrast")
plt.show()
```

Listing 27.11: Transform images by adjusting brightness and contrast

This shows the images as follows:

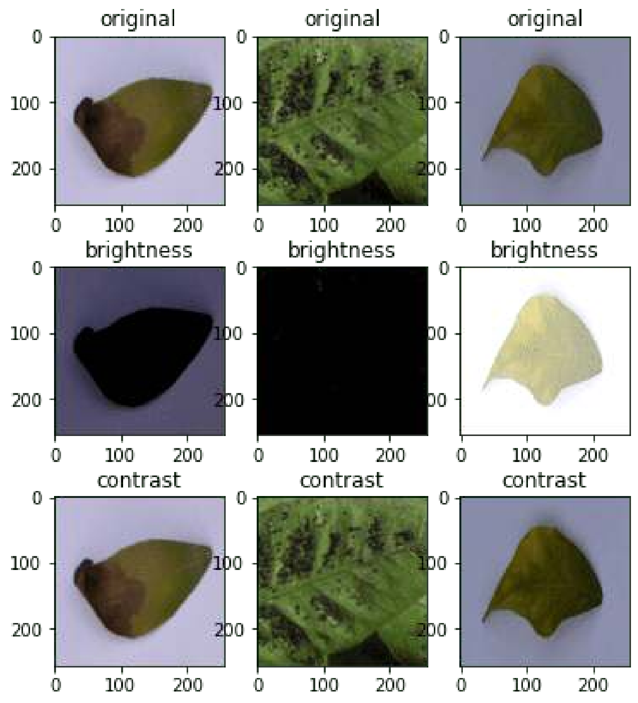


Figure 27.5: Example transformation from Listing 27.11

For completeness, below is the code to display the result of various augmentations:

```
from tensorflow.keras.utils import image_dataset_from_directory
import tensorflow as tf
import matplotlib.pyplot as plt
```

```
# use image_dataset_from_directory() to load images, with image size scaled to 256x256
PATH='.../Citrus/Leaves'  # modify to your path
ds = image_dataset_from_directory(PATH,
                                  validation_split=0.2, subset="training",
                                  image_size=(256,256), interpolation="mitchellcubic",
                                  crop_to_aspect_ratio=True,
                                  seed=42, shuffle=True, batch_size=32)

# Create preprocessing layers
out_height, out_width = 128,256
resize = tf.keras.layers.Resizing(out_height, out_width)
height = tf.keras.layers.RandomHeight(0.3)
width = tf.keras.layers.RandomWidth(0.3)
zoom = tf.keras.layers.RandomZoom(0.3)

flip = tf.keras.layers.RandomFlip("horizontal_and_vertical")
rotate = tf.keras.layers.RandomRotation(0.2)
crop = tf.keras.layers.RandomCrop(out_height, out_width)
translation = tf.keras.layers.RandomTranslation(height_factor=0.2, width_factor=0.2)

brightness = tf.keras.layers.RandomBrightness([-0.8,0.8])
contrast = tf.keras.layers.RandomContrast(0.2)

# Visualize images and augmentations
fig, ax = plt.subplots(5, 3, figsize=(6,14))
for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # resize
        ax[1][i].imshow(resize(images[i]).numpy().astype("uint8"))
        ax[1][i].set_title("resize")
        # height
        ax[2][i].imshow(height(images[i]).numpy().astype("uint8"))
        ax[2][i].set_title("height")
        # width
        ax[3][i].imshow(width(images[i]).numpy().astype("uint8"))
        ax[3][i].set_title("width")
        # zoom
        ax[4][i].imshow(zoom(images[i]).numpy().astype("uint8"))
        ax[4][i].set_title("zoom")
plt.show()

fig, ax = plt.subplots(5, 3, figsize=(6,14))
for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # flip
        ax[1][i].imshow(flip(images[i]).numpy().astype("uint8"))
        ax[1][i].set_title("flip")
        # crop
        ax[2][i].imshow(crop(images[i]).numpy().astype("uint8"))
        ax[2][i].set_title("crop")
        # translation
        ax[3][i].imshow(translation(images[i]).numpy().astype("uint8"))
        ax[3][i].set_title("translation")
        # rotate
        ax[4][i].imshow(rotate(images[i]).numpy().astype("uint8"))
        ax[4][i].set_title("rotate")
plt.show()

fig, ax = plt.subplots(3, 3, figsize=(6,7))
for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # brightness
        ax[1][i].imshow(brightness(images[i]).numpy().astype("uint8"))
        ax[1][i].set_title("brightness")
        # contrast
        ax[2][i].imshow(contrast(images[i]).numpy().astype("uint8"))
        ax[2][i].set_title("contrast")
plt.show()
```

Listing 27.12: Various augmentations using Keras preprocessing layers

Finally, it is important to point out that most neural network models can work better if the input images are scaled. While we usually use an 8-bit unsigned integer for the pixel values in an image (e.g., for display using `imshow()` as above), a neural network prefers the pixel values to be between 0 and 1 or between -1 and $+1$. This can be done with preprocessing layers too. Below is how you can update one of the examples above to add the scaling layer into the augmentation:

```
...
out_height, out_width = 128,256
resize = tf.keras.layers.Resizing(out_height, out_width)
rescale = tf.keras.layers.Rescaling(1/127.5, offset=-1)  # rescale pixel values to [-1,1]

def augment(image, label):
    return rescale(resize(image)), label

rescaled_resized_ds = ds.map(augment)

for image, label in rescaled_resized_ds:
   ...
```

Listing 27.13: Rescaling pixel values of an image

27.4 Using tf.image API for Augmentation

Besides the preprocessing layer, the `tf.image` module also provides some functions for augmentation. Unlike the preprocessing layer, these functions are intended to be used in a user-defined function and assigned to a dataset using `map()` as we saw above.

The functions provided by `tf.image` are not duplicates of the preprocessing layers, although there is some overlap. Below is an example of using the `tf.image` functions to resize and crop images:

```
...

fig, ax = plt.subplots(5, 3, figsize=(6,14))

for images, labels in ds.take(1):
    for i in range(3):
        # original
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # resize
        h = int(256 * tf.random.uniform([], minval=0.8, maxval=1.2))
        w = int(256 * tf.random.uniform([], minval=0.8, maxval=1.2))
        ax[1][i].imshow(tf.image.resize(images[i], [h,w]).numpy().astype("uint8"))
        ax[1][i].set_title("resize")
        # crop
        y, x, h, w = (128 * tf.random.uniform((4,))).numpy().astype("uint8")
        ax[2][i].imshow(tf.image.crop_to_bounding_box(images[i], y, x, h, w)
                        .numpy().astype("uint8"))
        ax[2][i].set_title("crop")
        # central crop
        x = tf.random.uniform([], minval=0.4, maxval=1.0)
        ax[3][i].imshow(tf.image.central_crop(images[i], x).numpy().astype("uint8"))
        ax[3][i].set_title("central crop")
        # crop to (h,w) at random offset
        h, w = (256 * tf.random.uniform((2,))).numpy().astype("uint8")
        seed = tf.random.uniform((2,), minval=0, maxval=65536).numpy().astype("int32")
        ax[4][i].imshow(tf.image.stateless_random_crop(images[i], [h,w,3], seed)
                        .numpy().astype("uint8"))
        ax[4][i].set_title("random crop")
plt.show()
```

Listing 27.14: Transform images by resizing and cropping using `tf.image` API

The output of the above code is in Figure 27.6.

While the display of images matches what you might expect from the code, the use of `tf.image` functions is quite different from that of the preprocessing layers. Every `tf.image` function is different. Therefore, you can see the `crop_to_bounding_box()` function takes pixel coordinates, but the `central_crop()` function assumes a fraction ratio as the argument.

These functions are also different in the way randomness is handled. Some of these functions do not assume random behavior. Therefore, the random resize should have the exact output size generated using a random number generator separately before calling the resize function. Some other functions, such as `stateless_random_crop()`, can do augmentation randomly, but a pair of random seeds in the `int32` needs to be specified explicitly.

To continue the example, there are the functions for flipping an image and extracting the Sobel edges:

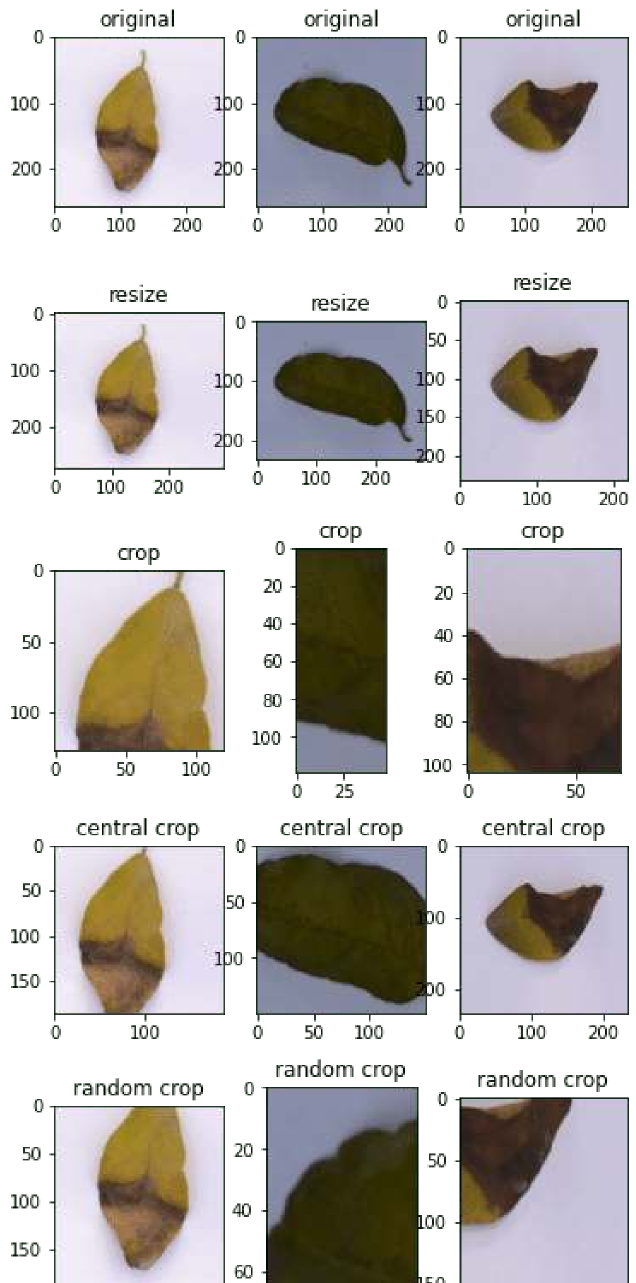


Figure 27.6: Example transformation from Listing 27.14

.

```
...
fig, ax = plt.subplots(5, 3, figsize=(6,14))

for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # flip
        seed = tf.random.uniform((2,), minval=0, maxval=65536).numpy().astype("int32")
        ax[1][i].imshow(tf.image.stateless_random_flip_left_right(images[i], seed)
                        .numpy().astype("uint8"))
        ax[1][i].set_title("flip left-right")
        # flip
        seed = tf.random.uniform((2,), minval=0, maxval=65536).numpy().astype("int32")
        ax[2][i].imshow(tf.image.stateless_random_flip_up_down(images[i], seed)
                        .numpy().astype("uint8"))
        ax[2][i].set_title("flip up-down")
        # sobel edge
        sobel = tf.image.sobel_edges(images[i:i+1])
        ax[3][i].imshow(sobel[0, ..., 0].numpy().astype("uint8"))
        ax[3][i].set_title("sobel y")
        # sobel edge
        ax[4][i].imshow(sobel[0, ..., 1].numpy().astype("uint8"))
        ax[4][i].set_title("sobel x")
plt.show()
```

Listing 27.15: Transform images by flipping and sobel transform

This shows the image in Figure 27.7.

And the following are the functions to manipulate the brightness, contrast, and colors:

```
...
fig, ax = plt.subplots(5, 3, figsize=(6,14))

for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # brightness
        seed = tf.random.uniform((2,), minval=0, maxval=65536).numpy().astype("int32")
        ax[1][i].imshow(tf.image.stateless_random_brightness(images[i], 0.3, seed)
                        .numpy().astype("uint8"))
        ax[1][i].set_title("brightness")
        # contrast
        ax[2][i].imshow(tf.image.stateless_random_contrast(images[i], 0.7, 1.3, seed)
                        .numpy().astype("uint8"))
        ax[2][i].set_title("contrast")
        # saturation
        ax[3][i].imshow(tf.image.stateless_random_saturation(images[i], 0.7, 1.3, seed)
                        .numpy().astype("uint8"))
        ax[3][i].set_title("saturation")
        # hue
        ax[4][i].imshow(tf.image.stateless_random_hue(images[i], 0.3, seed)
```

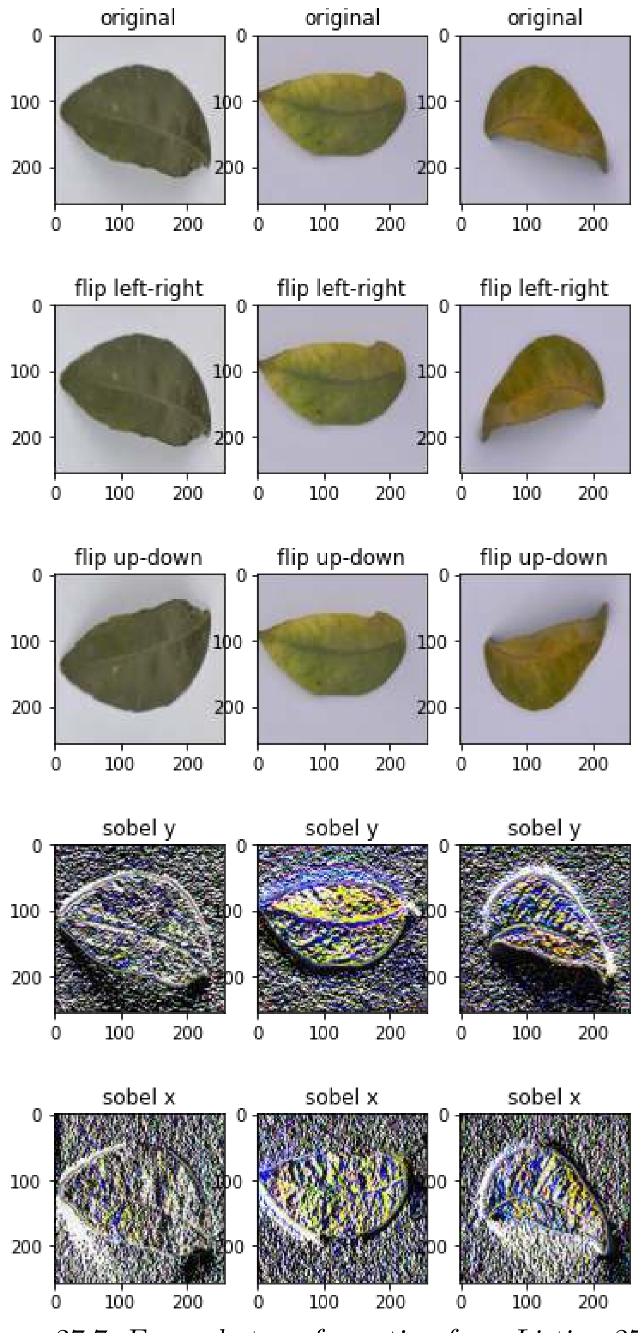


Figure 27.7: Example transformation from Listing 27.15

```
                         .numpy().astype("uint8"))
          ax[4][i].set_title("hue")
plt.show()
```

Listing 27.16: Transform images by adjusting brightness, contrast, saturation, and hue

This code shows the image in Figure 27.8.

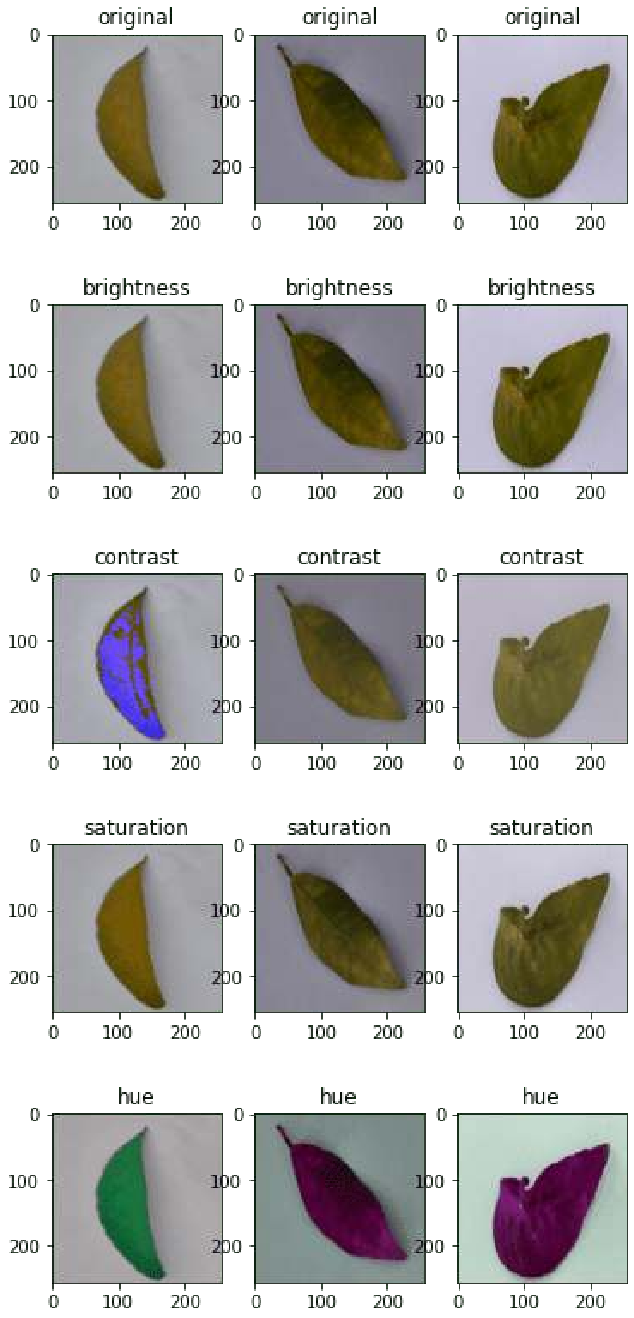


Figure 27.8: Example transformation from Listing 27.16

Below is the complete code to display all of the above:

```
from tensorflow.keras.utils import image_dataset_from_directory
import tensorflow as tf
import matplotlib.pyplot as plt

# use image_dataset_from_directory() to load images, with image size scaled to 256x256
PATH='.../Citrus/Leaves'  # modify to your path
ds = image_dataset_from_directory(PATH,
                                  validation_split=0.2, subset="training",
                                  image_size=(256,256), interpolation="mitchellcubic",
                                  crop_to_aspect_ratio=True,
                                  seed=42, shuffle=True, batch_size=32)

# Visualize tf.image augmentations

fig, ax = plt.subplots(5, 3, figsize=(6,14))
for images, labels in ds.take(1):
    for i in range(3):
        # original
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # resize
        h = int(256 * tf.random.uniform([], minval=0.8, maxval=1.2))
        w = int(256 * tf.random.uniform([], minval=0.8, maxval=1.2))
        ax[1][i].imshow(tf.image.resize(images[i], [h,w]).numpy().astype("uint8"))
        ax[1][i].set_title("resize")
        # crop
        y, x, h, w = (128 * tf.random.uniform((4,))).numpy().astype("uint8")
        ax[2][i].imshow(tf.image.crop_to_bounding_box(images[i], y, x, h, w)
                        .numpy().astype("uint8"))
        ax[2][i].set_title("crop")
        # central crop
        x = tf.random.uniform([], minval=0.4, maxval=1.0)
        ax[3][i].imshow(tf.image.central_crop(images[i], x).numpy().astype("uint8"))
        ax[3][i].set_title("central crop")
        # crop to (h,w) at random offset
        h, w = (256 * tf.random.uniform((2,))).numpy().astype("uint8")
        seed = tf.random.uniform((2,), minval=0, maxval=65536).numpy().astype("int32")
        ax[4][i].imshow(tf.image.stateless_random_crop(images[i], [h,w,3], seed)
                        .numpy().astype("uint8"))
        ax[4][i].set_title("random crop")
plt.show()

fig, ax = plt.subplots(5, 3, figsize=(6,14))
for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # flip
        seed = tf.random.uniform((2,), minval=0, maxval=65536).numpy().astype("int32")
        ax[1][i].imshow(tf.image.stateless_random_flip_left_right(images[i], seed)
                        .numpy().astype("uint8"))
        ax[1][i].set_title("flip left-right")
        # flip
        seed = tf.random.uniform((2,), minval=0, maxval=65536).numpy().astype("int32")
        ax[2][i].imshow(tf.image.stateless_random_flip_up_down(images[i], seed)
                        .numpy().astype("uint8"))
        ax[2][i].set_title("flip up-down")
        # sobel edge
        sobel = tf.image.sobel_edges(images[i:i+1])
        ax[3][i].imshow(sobel[0, ..., 0].numpy().astype("uint8"))
        ax[3][i].set_title("sobel y")
        # sobel edge
        ax[4][i].imshow(sobel[0, ..., 1].numpy().astype("uint8"))
        ax[4][i].set_title("sobel x")
plt.show()

fig, ax = plt.subplots(5, 3, figsize=(6,14))
for images, labels in ds.take(1):
    for i in range(3):
        ax[0][i].imshow(images[i].numpy().astype("uint8"))
        ax[0][i].set_title("original")
        # brightness
        seed = tf.random.uniform((2,), minval=0, maxval=65536).numpy().astype("int32")
        ax[1][i].imshow(tf.image.stateless_random_brightness(images[i], 0.3, seed)
                        .numpy().astype("uint8"))
        ax[1][i].set_title("brightness")
        # contrast
        ax[2][i].imshow(tf.image.stateless_random_contrast(images[i], 0.7, 1.3, seed)
                        .numpy().astype("uint8"))
        ax[2][i].set_title("contrast")
        # saturation
        ax[3][i].imshow(tf.image.stateless_random_saturation(images[i], 0.7, 1.3, seed)
                        .numpy().astype("uint8"))
        ax[3][i].set_title("saturation")
        # hue
        ax[4][i].imshow(tf.image.stateless_random_hue(images[i], 0.3, seed)
                        .numpy().astype("uint8"))
        ax[4][i].set_title("hue")
plt.show()
```

Listing 27.17: Various augmentations using `tf.image` API

These augmentation functions should be enough for most uses. But if you have some specific ideas on augmentation, you would probably need a better image processing library. OpenCV and Pillow are common but powerful libraries that allow you to transform images better.

27.5 Using Preprocessing Layers in Neural Networks

You used the Keras preprocessing layers as functions in the examples above. But they can also be used as layers in a neural network. It is trivial to use. Below is an example of how you can incorporate a preprocessing layer into a classification network and train it using a dataset:

```python
from tensorflow.keras.utils import image_dataset_from_directory
import tensorflow as tf
import matplotlib.pyplot as plt

# use image_dataset_from_directory() to load images, with image size scaled to 256x256
PATH='.../Citrus/Leaves'  # modify to your path
ds = image_dataset_from_directory(PATH,
                                  validation_split=0.2, subset="training",
                                  image_size=(256,256), interpolation="mitchellcubic",
                                  crop_to_aspect_ratio=True,
                                  seed=42, shuffle=True, batch_size=32)

AUTOTUNE = tf.data.AUTOTUNE
ds = ds.cache().prefetch(buffer_size=AUTOTUNE)

num_classes = 5
model = tf.keras.Sequential([
  tf.keras.layers.RandomFlip("horizontal_and_vertical"),
  tf.keras.layers.RandomRotation(0.2),
  tf.keras.layers.Rescaling(1/127.0, offset=-1),
  tf.keras.layers.Conv2D(32, 3, activation='relu'),
  tf.keras.layers.MaxPooling2D(),
  tf.keras.layers.Conv2D(32, 3, activation='relu'),
  tf.keras.layers.MaxPooling2D(),
  tf.keras.layers.Conv2D(32, 3, activation='relu'),
  tf.keras.layers.MaxPooling2D(),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(num_classes)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

model.fit(ds, epochs=3)
```

Listing 27.18: Using preprocessing layer as part of a neural network

Running this code gives the following output:

```
Found 609 files belonging to 5 classes.
Using 488 files for training.
Epoch 1/3
16/16 [==============================] - 5s 253ms/step - loss: 1.4114 - accuracy: 0.4283
Epoch 2/3
16/16 [==============================] - 4s 259ms/step - loss: 0.8101 - accuracy: 0.6475
Epoch 3/3
16/16 [==============================] - 4s 267ms/step - loss: 0.7015 - accuracy: 0.7111
```

Output 27.4: Example output on training a network with preprocessing layers

In the code above, you created the dataset with `cache()` and `prefetch()`. This is a performance technique to allow the dataset to prepare data asynchronously while the neural network is trained. This would be significant if the dataset has some other augmentation assigned using the `map()` function.

You will see some improvement in accuracy if you remove the `RandomFlip` and `RandomRotation` layers because you make the problem easier. However, as you want the network to predict well on a wide variation of image quality and properties, using augmentation can help you resulting network become more powerful.

27.6 Further Reading

Below are documentations from TensorFlow that are related to the examples above:

APIs

tf.data.Dataset. TensorFlow API.
https://www.tensorflow.org/api_docs/python/tf/data/Dataset
Module: tf.image. TensorFlow API.
https://www.tensorflow.org/api_docs/python/tf/image

Articles

Citrus Leaves. TensorFlow Datasets.
https://www.tensorflow.org/datasets/catalog/citrus_leaves
Load and preprocess images. TensorFlow Tutorials.
https://www.tensorflow.org/tutorials/load_data/images
Data augmentation. TensorFlow Tutorials.
https://www.tensorflow.org/tutorials/images/data_augmentation
Better performance with the tf.data API. TensorFlow guide.
https://www.tensorflow.org/guide/data_performance

27.7 Summary

In this chapter, you have seen how you can use the `tf.data` dataset with image augmentation functions from Keras and TensorFlow.

Specifically, you learned:

- ▷ How to use the preprocessing layers from Keras, both as a function and as part of a neural network
- ▷ How to create your own image augmentation function and apply it to the dataset using the `map()` function
- ▷ How to use the functions provided by the `tf.image` module for image augmentation

In the next chapter, you will see an example of doing classification using CNN and how a deeper network can improve accuracy.

28 Project: Object Classification in Photographs

A difficult problem where traditional neural networks fall down is called object recognition. It is where a model is able to identify objects in images. In this chapter, you will discover how to develop and evaluate deep learning models for object recognition in Keras. After completing this chapter, you will know:

- About the CIFAR-10 object classification dataset and how to load and use it in Keras
- How to create a simple Convolutional Neural Network for object recognition
- How to lift performance by creating deeper Convolutional Neural Networks

Let's get started.

Note: You may want to speed up the computation for this chapter by using GPU rather than CPU hardware, such as the process described in Appendix C. This is a suggestion, not a requirement. The code will work just fine on the CPU.

Overview

This chapter is divided into five sections; they are:

- The CIFAR-10 Dataset
- Loading the CIFAR-10 Dataset in Keras
- Simple Convolutional Neural Network for CIFAR-10
- Larger Convolutional Neural Network for CIFAR-10
- Extensions To Improve Model Performance

28.1 The CIFAR-10 Dataset

The problem of automatically classifying objects in photographs is difficult because of the nearly infinite number of permutations of objects, positions, lighting, and so on. It's a tough problem. This is a well-studied problem in computer vision and, more recently, an important demonstration of the capability of deep learning. A standard computer vision and deep learning dataset for this problem was developed by the Canadian Institute for Advanced Research (CIFAR).

The CIFAR-10 dataset consists of 60,000 photos divided into 10 classes (hence the name CIFAR-10). Classes include common objects such as airplanes, automobiles, birds, cats, and so on. The dataset is split in a standard way, where 50,000 images are used for training a model and the remaining 10,000 for evaluating its performance. The photos are in color with red, green, and blue channels, but are small, measuring 32×32 pixel squares.

State-of-the-art results can be achieved using very large convolutional neural networks. You can learn about state-of-the-art results on CIFAR-10 on Rodrigo Benenson's webpage. Model performance is reported in classification accuracy, with very good performance above 90%, with human performance on the problem at 94% and state-of-the-art results at 96% at the time of writing.

28.2 Loading The CIFAR-10 Dataset in Keras

The CIFAR-10 dataset can easily be loaded in Keras. Keras has the facility to automatically download standard datasets like CIFAR-10 and store them in the `~/.keras/datasets` directory using the `cifar10.load_data()` function. This dataset is large at 163 megabytes, so it may take a few minutes to download. Once downloaded, subsequent calls to the function will load the dataset ready for use.

The dataset is stored as *pickled* training and test sets, ready for use in Keras. Each image is represented as a three-dimensional matrix, with dimensions for color (red, green, blue), width, and height. We can plot images directly using matplotlib.

```python
# Plot ad hoc CIFAR10 instances
from tensorflow.keras.datasets import cifar10
import matplotlib.pyplot as plt
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# create a grid of 3x3 images
for i in range(0, 9):
    plt.subplot(330 + 1 + i)
    plt.imshow(X_train[i])
# show the plot
plt.show()
```

Listing 28.1: Load and plot sample CIFAR-10 images

Running the code creates a 3×3 plot of photographs. The images have been scaled up from their small 32×32 size, but you can clearly see trucks, horses, and cars. You can also see some distortion in the images that have been forced to the square aspect ratio.

Figure 28.1: Small sample of CIFAR-10 images

28.3 Simple Convolutional Neural Network for CIFAR-10

The CIFAR-10 problem is best solved using a convolutional neural network (CNN). You can quickly start by importing all the classes and functions you will need in this example.

```
# Simple CNN model for CIFAR-10
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.utils import to_categorical
...
```

Listing 28.2: Load classes and functions

Next, you can load the CIFAR-10 dataset.

```
...
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

Listing 28.3: Load the CIFAR-10 dataset

The pixel values range from 0 to 255 for each of the red, green, and blue channels. It is good practice to work with normalized data. Because the input values are well understood, you can easily normalize to the range 0 to 1 by dividing each value by the maximum observation, which is 255. Note that the data is loaded as integers, so you must cast it to floating point values in order to perform the division.

```
...
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
```

Listing 28.4: Normalize the CIFAR-10 dataset

The output variables are defined as a vector of integers from 0 to 1 for each class. You can use a one-hot encoding to transform them into a binary matrix to best model the classification problem. There are ten classes for this problem, so you can expect the binary matrix to have a width of 10.

```
...
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
```

Listing 28.5: One-hot encode the output variable

Let's start by defining a simple CNN structure as a baseline and evaluate how well it performs on the problem. You will use a structure with two convolutional layers followed by max pooling and a flattening out of the network to fully connected layers to make predictions. You baseline network structure can be summarized as follows:

1. Convolutional input layer, 32 feature maps with a size of 3×3, a rectifier activation function, and a weight constraint of max norm set to 3
2. Dropout set to 20%
3. Convolutional layer, 32 feature maps with a size of 3×3, a rectifier activation function, and a weight constraint of max norm set to 3
4. Max Pool layer with the size 2×2
5. Flatten layer
6. Fully connected layer with 512 units and a rectifier activation function
7. Dropout set to 50%
8. Fully connected output layer with 10 units and a softmax activation function

A logarithmic loss function is used with the stochastic gradient descent optimization algorithm configured with a large momentum and weight decay, starting with a learning rate of 0.01. A visualization of the network structure is provided below.

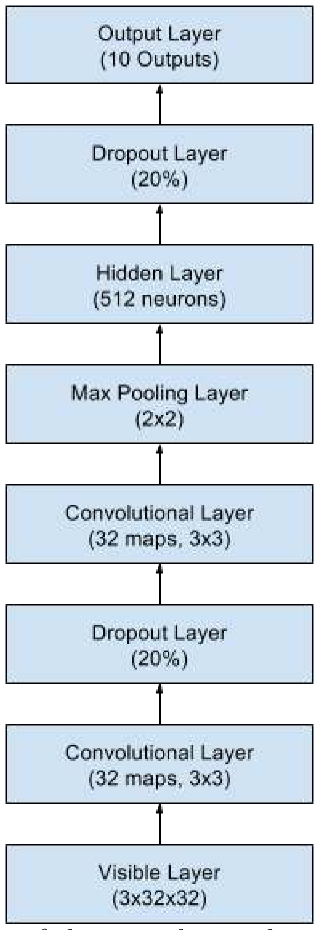


Figure 28.2: Summary of the convolutional neural network structure

```
...
# Create the model
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3), padding='same',
                 activation='relu', kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same',
                 kernel_constraint=MaxNorm(3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu', kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
# Compile model
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(learning_rate=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
model.summary()
```

Listing 28.6: Define and compile the CNN model

You fit this model with 25 epochs and a batch size of 32. A small number of epochs was chosen to help keep this chapter moving. Usually the number of epochs would be one or two orders of magnitude larger for this problem. Once the model is fit, you evaluate it on the test dataset and print out the classification accuracy.

```
...
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=epochs, batch_size=32)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 28.7: Evaluate the accuracy of the CNN model

Tying this all together, the complete example is listed below.

```
# Simple CNN model for the CIFAR-10 Dataset
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.utils import to_categorical
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
# Create the model
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3), padding='same',
          activation='relu', kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same',
          kernel_constraint=MaxNorm(3)))
model.add(MaxPooling2D())
model.add(Flatten())
model.add(Dense(512, activation='relu', kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
# Compile model
epochs = 25
```

```
lrate = 0.01
decay = lrate/epochs
sgd = SGD(learning_rate=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
model.summary(line_length=72)
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=epochs, batch_size=32)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 28.8: CNN model for CIFAR-10 problem

Running this example provides the results below. First, the network structure is summarized, which confirms the design was implemented correctly.

```
Model: "sequential"
________________________________________________________________________
 Layer (type)                Output Shape              Param #
========================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

 dropout (Dropout)           (None, 32, 32, 32)        0

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d (MaxPooling2D)  (None, 16, 16, 32)      0

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 512)               4194816

 dropout_1 (Dropout)         (None, 512)               0

 dense_1 (Dense)             (None, 10)                5130

========================================================================
Total params: 4,210,090
Trainable params: 4,210,090
Non-trainable params: 0
________________________________________________________________________
```

Output 28.1: Network structure of the CNN model for CIFAR-10 problem

The classification accuracy and loss is printed each epoch on both the training and test datasets.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

The model is evaluated on the test set and achieves an accuracy of 70.5%, which is not excellent.

```
...
Epoch 20/25
1563/1563 [==============================] - 34s 22ms/step - loss: 0.3001 - accuracy: 0.8944 - val_l
Epoch 21/25
1563/1563 [==============================] - 35s 23ms/step - loss: 0.2783 - accuracy: 0.9021 - val_l
Epoch 22/25
1563/1563 [==============================] - 35s 22ms/step - loss: 0.2623 - accuracy: 0.9084 - val_l
Epoch 23/25
1563/1563 [==============================] - 33s 21ms/step - loss: 0.2536 - accuracy: 0.9104 - val_l
Epoch 24/25
1563/1563 [==============================] - 34s 22ms/step - loss: 0.2383 - accuracy: 0.9180 - val_l
Epoch 25/25
1563/1563 [==============================] - 37s 24ms/step - loss: 0.2245 - accuracy: 0.9219 - val_l
Accuracy: 70.50%
```

Output 28.2: Sample output for the CNN model

You can improve the accuracy significantly by creating a much deeper network. This is what you will look at in the next section.

28.4 Larger Convolutional Neural Network for CIFAR-10

You have seen that a simple CNN performs poorly on this complex problem. In this section, you will look at scaling up the size and complexity of your model. Let's design a deep version of the simple CNN above. You can introduce an additional round of convolutions with many more feature maps. You will use the same pattern of convolutional, dropout, convolutional and max pooling layers.

This pattern will be repeated three times with 32, 64, and 128 feature maps. The effect is an increasing number of feature maps with a smaller and smaller size given the max pooling layers. Finally, an additional and larger `Dense` layer will be used at the output end of the network in an attempt to better translate the large number of feature maps to class values. A summary of the new network architecture is as follows:

1. Convolutional input layer, 32 feature maps with a size of 3×3 and a rectifier activation function.
2. Dropout layer at 20%.
3. Convolutional layer, 32 feature maps with a size of 3×3 and a rectifier activation function.
4. Max Pool layer with size 2×2.
5. Convolutional layer, 64 feature maps with a size of 3×3 and a rectifier activation function.
6. Dropout layer at 20%.
7. Convolutional layer, 64 feature maps with a size of 3×3 and a rectifier activation function.
8. Max Pool layer with size 2×2.
9. Convolutional layer, 128 feature maps with a size of 3×3 and a rectifier activation function.

10. Dropout layer at 20%.
11. Convolutional layer, 128 feature maps with a size of 3×3 and a rectifier activation function.
12. Max Pool layer with size 2×2.
13. Flatten layer.
14. Dropout layer at 20%.
15. Fully connected layer with 1,024 units and a rectifier activation function.
16. Dropout layer at 20%.
17. Fully connected layer with 512 units and a rectifier activation function.
18. Dropout layer at 20%.
19. Fully connected output layer with 10 units and a softmax activation function.

You can very easily define this network topology in Keras as follows:

```python
...
# Create the model
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D())
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D())
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D())
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu', kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu', kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
# Compile model
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(learning_rate=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
model.summary()
...
```

Listing 28.9: Defining a larger CNN for CIFAR-10

You can fit and evaluate this model using the same procedure from above and the same number of epochs but a larger batch size of 64, found through some minor experimentation.

```
...
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=epochs, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 28.10: Fitting the larger CNN with a larger batch size

Tying this all together, the complete example is listed below.

```
# Large CNN model for the CIFAR-10 Dataset
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.utils import to_categorical
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
# one-hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
num_classes = y_test.shape[1]
# Create the model
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D())
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D())
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Dropout(0.2))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D())
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu', kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu', kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
```

```
# Compile model
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(learning_rate=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
model.summary()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=epochs, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 28.11: Large CNN model for CIFAR-10 problem

Running this example prints the classification accuracy and loss on the training and test datasets for each epoch.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

The estimate of classification accuracy for the final model is 79.5% which is nine points better than our simpler model.

```
...
Epoch 20/25
782/782 [==============================] - 50s 64ms/step - loss: 0.4949 - accuracy: 0.8237 - val_los
Epoch 21/25
782/782 [==============================] - 51s 65ms/step - loss: 0.4794 - accuracy: 0.8308 - val_los
Epoch 22/25
782/782 [==============================] - 50s 64ms/step - loss: 0.4660 - accuracy: 0.8347 - val_los
Epoch 23/25
782/782 [==============================] - 50s 64ms/step - loss: 0.4523 - accuracy: 0.8395 - val_los
Epoch 24/25
782/782 [==============================] - 50s 64ms/step - loss: 0.4344 - accuracy: 0.8454 - val_los
Epoch 25/25
782/782 [==============================] - 50s 64ms/step - loss: 0.4231 - accuracy: 0.8487 - val_los
Accuracy: 79.50%
```

Output 28.3: Sample output for fitting the larger CNN model

28.5 Extensions to Improve Model Performance

You have achieved good results on this very difficult problem, but you are still a good way from achieving world-class results. Below are some ideas that you can try to extend upon the model and improve model performance.

- **Train for More Epochs**. Each model was trained for a very small number of epochs, 25. It is common to train large convolutional neural networks for hundreds or thousands of epochs. You should expect performance gains can be achieved by significantly raising the number of training epochs.

- **Image Data Augmentation**. The objects in the image vary in their position. Another boost in model performance can likely be achieved by using some data augmentation. Methods such as standardization, random shifts, and horizontal image flips may be beneficial.
- **Deeper Network Topology**. The larger network presented is deep, but larger networks could be designed for the problem. This may involve more feature maps closer to the input and perhaps less aggressive pooling. Additionally, standard convolutional network topologies that have been shown useful may be adopted and evaluated on the problem.

28.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Articles

Alex Krizhevsky. *The CIFAR-10 dataset.*
`https://www.cs.toronto.edu/~kriz/cifar.html`

Rodrigo Benenson. *What is the class of this image?* Classification datasets results, 2016.
`https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`

28.7 Summary

In this chapter, you discovered how to create deep learning models in Keras for object recognition in photographs. After working through this chapter, you learned:

- About the CIFAR-10 dataset and how to load it in Keras and plot ad hoc examples from the dataset
- How to train and evaluate a simple Convolutional Neural Network on the problem
- How to expand a simple convolutional neural network into a deep convolutional neural network in order to boost performance on the difficult problem
- How to use data augmentation to get a further boost on the difficult object recognition problem

In the next chapter, you will see that CNN can also be applied to non-image data.

29 Project: Predict Sentiment from Movie Reviews

Sentiment analysis is a natural language processing problem where text is understood, and the underlying intent is predicted. In this chapter, you will discover how you can predict the sentiment of movie reviews as either positive or negative in Python using the Keras deep learning library. After completing this chapter, you will know:

- ▷ About the IMDB sentiment analysis problem for natural language processing and how to load it in Keras
- ▷ How to use word embedding in Keras for natural language problems
- ▷ How to develop and evaluate a multilayer perceptron model for the IMDB problem
- ▷ How to develop a one-dimensional convolutional neural network model for the IMDB problem

Let's get started.

Overview

This chapter is divided into five sections; they are:

- ▷ IMDB Movie Review Sentiment Problem
- ▷ Load the IMDB Dataset With Keras
- ▷ Word Embeddings
- ▷ Simple Multilayer Perceptron Model for the IMDB Dataset
- ▷ One-Dimensional Convolutional Neural Network Model for the IMDB Dataset

29.1 IMDB Movie Review Sentiment Problem

The dataset is the *Large Movie Review Dataset*, often referred to as the IMDB dataset. The IMDB dataset contains 25,000 highly polar movie reviews (good or bad) for training and the same amount again for testing. The problem is to determine whether a given movie review has a positive or negative sentiment.

The data was collected by Stanford researchers and used in Maas et al., "Learning Word Vectors for Sentiment Analysis" where a split of 50-50 of the data was used for training and test. An accuracy of 88.89% was achieved.

29.2 Load the IMDB Dataset with Keras

Keras provides built-in access to the IMDB dataset. The `tf.keras.datasets.imdb.load_data()` allows you to load the dataset in a format that is ready for use in neural networks and deep learning models. The words have been replaced by integers that indicate the absolute popularity of the word in the dataset. The sentences in each review are therefore comprised of a sequence of integers.

Calling `imdb.load_data()` the first time will download the IMDB dataset to your computer and store it in your home directory under `~/.keras/datasets/imdb.npz` as a 17MB file. Usefully, the `imdb.load_data()` provides additional arguments, including the number of top words to load (where words with a lower integer are marked as zero in the returned data), the number of top words to skip (to avoid the repeated use of *the*), and the maximum length of reviews to support. Let's load the dataset and calculate some properties of it. You will start by loading some libraries and the entire IMDB dataset as a training dataset.

```python
import numpy as np
from tensorflow.keras.datasets import imdb
import matplotlib.pyplot as plt
# load the dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data()
X = np.concatenate((X_train, X_test), axis=0)
y = np.concatenate((y_train, y_test), axis=0)
...
```

Listing 29.1: Load the IMDB dataset

Next, you can display the shape of the training dataset.

```python
...
# summarize size
print("Training data: ")
print(X.shape)
print(y.shape)
```

Listing 29.2: Display the shape of the IMDB dataset

Running this snippet, you can see that there are 50,000 records.

```
Training data:
(50000,)
(50000,)
```

Output 29.1: Output of the shape of the IMDB dataset

You can also print the unique class values.

```
...
# Summarize number of classes
print("Classes: ")
print(np.unique(y))
```

Listing 29.3: Display the classes in IMDB dataset

You can see it is a binary classification problem for good and bad sentiment in the review.

```
Classes:
[0 1]
```

Output 29.2: Output of the classes of the IMDB dataset

Next, you can get an idea of the total number of unique words in the dataset.

```
...
# Summarize number of words
print("Number of words: ")
print(len(np.unique(np.hstack(X))))
```

Listing 29.4: Display the number of unique words in the IMDB dataset

Interestingly, you can see that there are just under 100,000 words across the entire dataset.

```
Number of words:
88585
```

Output 29.3: Output of the classes of unique words in the IMDB dataset

Finally, you can get an idea of the average review length.

```
...
# Summarize review length
print("Review length: ")
result = [len(x) for x in X]
print("Mean %.2f words (%f)" % (np.mean(result), np.std(result)))
# plot review length
plt.subplot(121)
plt.boxplot(result)
plt.subplot(122)
plt.hist(result)
plt.show()
```

Listing 29.5: Plot the distriobution of review lengths

You can see that the average review has just under 300 words with a standard deviation of just over 200 words.

```
Review length:
Mean 234.76 words (172.911495)
```

Output 29.4: Output of the distribution summary for review length

Looking at the box and whisker plot and the histogram for the review lengths in words, you can see an exponential distribution that you can probably cover the mass of the distribution with a clipped length of 400 to 500 words.

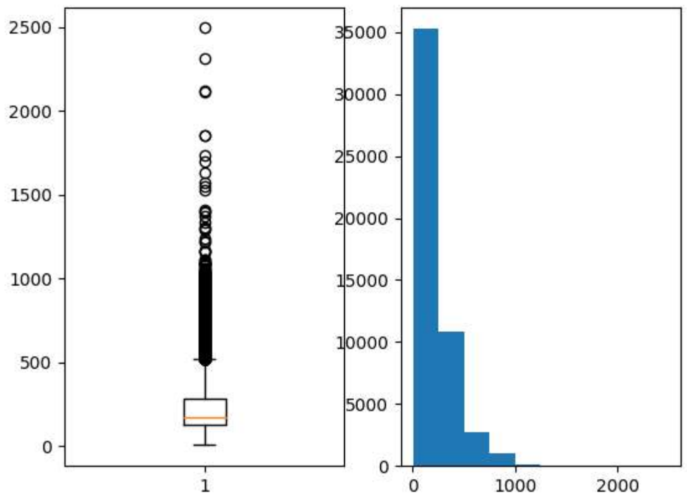


Figure 29.1: Review length in words for IMDB dataset

29.3 Word Embeddings

A recent breakthrough in the field of natural language processing is called word embedding. This technique is where words are encoded as real-valued vectors in a high-dimensional space, where the similarity between words in terms of meaning translates to closeness in the vector space. Discrete words are mapped to vectors of continuous numbers. This is useful when working with natural language problems with neural networks and deep learning models as they require numbers as input values.

Keras provides a convenient way to convert positive integer representations of words into a word embedding by an `Embedding` layer. The layer takes arguments that define the mapping, including the maximum number of expected words, also called the vocabulary size (e.g., the largest integer value that will be seen as an input). The layer also allows you to specify the dimensionality for each word vector, called the output dimension.

You want to use a word embedding representation for the IMDB dataset. Let's say that you are only interested in the first 5,000 most used words in the dataset. Therefore, your vocabulary size will be 5,000. You can choose to use a 32-dimensional vector to represent each word. Finally, you may choose to cap the maximum review length at 500 words, truncating reviews longer than that and padding reviews shorter than that with 0 values. You will load the IMDB dataset as follows:

```
...
imdb.load_data(num_words=5000)
```

Listing 29.6: Load only the top 5000 words in the IMDB review

You will then use the Keras utility to truncate or pad the dataset to a length of 500 for each observation using the `sequence.pad_sequences()` function.

```
...
X_train = sequence.pad_sequences(X_train, maxlen=500)
X_test = sequence.pad_sequences(X_test, maxlen=500)
```

Listing 29.7: Pad reviews in the IMDB dataset

Finally, later on, the first layer of your model would be a word embedding layer created using the `Embedding` class as follows:

```
...
Embedding(5000, 32, input_length=500)
```

Listing 29.8: Define a word embedding representation

The output of this first layer would be a matrix with the size 32×500 for a given movie review training or test pattern in integer format. Now that you know how to load the IMDB dataset in Keras and how to use a word embedding representation for it, let's develop and evaluate some models.

29.4 Simple Multilayer Perceptron Model for the IMDB Dataset

You can start by developing a simple multilayer perceptron model with a single hidden layer. The word embedding representation is a true innovation, and you will demonstrate what would have been considered world-class results in 2011 with a relatively simple neural network. Let's start by importing the classes and functions required for this model and initializing the random number generator to a constant value to ensure you can easily reproduce the results.

```
# MLP for the IMDB problem
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence

...
```

Listing 29.9: Load classes and functions

Next, you will load the IMDB dataset. You will simplify the dataset as discussed during the section on word embeddings — only the top 5,000 words will be loaded. You will also use a 50/50 split of the dataset into training and test sets. This is a good standard split methodology.

```
...
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
```

Listing 29.10: Load and split the IMDB dataset

You will bound reviews at 500 words, truncating longer reviews and zero-padding shorter ones.

```
...
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
```

Listing 29.11: Pad IMDB reviews to a fixed length

Now, you can create your model. You will use an `Embedding` layer as the input layer, setting the vocabulary to 5,000, the word vector size to 32 dimensions, and the `input_length` to 500. The output of this first layer will be a 32×500-sized matrix, as discussed in the previous section. You will flatten the `Embedded` layers' output to one dimension, then use one dense hidden layer of 250 units with a rectifier activation function. The output layer has one neuron and will use a sigmoid activation to output values of 0 and 1 as predictions. The model uses logarithmic loss and is optimized using the efficient ADAM optimization procedure.

```
...
# create the model
model = Sequential()
model.add(Embedding(top_words, 32, input_length=max_words))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Listing 29.12: Define a multilayer perceptron model

You can fit the model and use the test set as validation while training. This model overfits very quickly, so you will use very few training epochs, in this case, just 2. There is a lot of data, so you will use a batch size of 128. After the model is trained, you will evaluate its accuracy on the test dataset.

```
...
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=2, batch_size=128, verbose=1)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 29.13: Fit and evaluate the multilayer pereceptron model

Tying all of this together, the complete code listing is provided below.

```
# MLP for the IMDB problem
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
# create the model
model = Sequential()
model.add(Embedding(top_words, 32, input_length=max_words))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=2, batch_size=128, verbose=1)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 29.14: Multilayer perceptron model for the IMDB dataset

Running this example fits the model and summarizes the estimated performance.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

You can see that this very simple model achieves a score of 87%, which is in the neighborhood of the original paper, with very minimal effort.

```
Epoch 1/2
196/196 - 4s - loss: 0.5579 - accuracy: 0.6664 - val_loss: 0.3060 - val_accuracy: 0.8700 - 4s/epoch
Epoch 2/2
196/196 - 4s - loss: 0.2108 - accuracy: 0.9165 - val_loss: 0.3006 - val_accuracy: 0.8731 - 4s/epoch
Accuracy: 87.31%
```

Output 29.5: Output from evaluating the multilayer perceptron model

You can likely do better if you trained this network, perhaps using a larger embedding and adding more hidden layers.

Let's try a different network type.

29.5 One-Dimensional Convolutional Neural Network Model for the IMDB Dataset

Convolutional neural networks were designed to honor the spatial structure in image data while being robust to the position and orientation of learned objects in the scene. This same principle can be used on sequences, such as the one-dimensional sequence of words in a movie review. The same properties that make the CNN model attractive for learning to recognize objects in images can help to learn structure in paragraphs of words, namely the techniques invariance to the specific position of features.

Keras supports one-dimensional convolutions and pooling by the `Conv1D` and `MaxPooling1D` classes, respectively. Again, let's import the classes and functions needed for this example and initialize your random number generator to a constant value so that you can easily reproduce the results.

```
# CNN for the IMDB problem
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import MaxPooling1D
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
```

Listing 29.15: Import classes and functions

You can also load and prepare the IMDB dataset as you did before.

```
...
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
# pad dataset to a maximum review length in words
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
```

Listing 29.16: Load, split, and pad IMDB dataset

You can now define your convolutional neural network model. This time, after the `Embedding` input layer, insert a `Conv1D` layer. This convolutional layer has 32 feature maps and reads embedded word representations' three vector elements of the word embedding at a time. The convolutional layer is followed by a `MaxPooling1D` layer with a length and stride of 2 that halves the size of the feature maps from the convolutional layer. The rest of the network is the same as the neural network above.

```
...
# create the model
model = Sequential()
```

```
model.add(Embedding(top_words, 32, input_length=max_words))
model.add(Conv1D(32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Listing 29.17: Define the CNN model

You will also fit the network the same as before.

```
...
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=2, batch_size=128, verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 29.18: Fit and evaluate the CNN model

Tying all of this together, the complete code listing is provided below.

```
# CNN for the IMDB problem
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import MaxPooling1D
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
# pad dataset to a maximum review length in words
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
# create the model
model = Sequential()
model.add(Embedding(top_words, 32, input_length=max_words))
model.add(Conv1D(32, 3, padding='same', activation='relu'))
model.add(MaxPooling1D())
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test),
          epochs=2, batch_size=128, verbose=2)
```

```
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 29.19: CNN model for the IMDB dataset

Running the example, you are first presented with a summary of the network structure (not shown here). You can see your convolutional layer preserves the dimensionality of your `Embedding` input layer of 32-dimensional input with a maximum of 500 words. The pooling layer compresses this representation by halving it.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example offers a slight but welcome improvement over the neural network model above with an accuracy of 87%.

```
Epoch 1/2
196/196 - 5s - loss: 0.4661 - accuracy: 0.7467 - val_loss: 0.2763 - val_accuracy: 0.8860 - 5s/epoch
Epoch 2/2
196/196 - 5s - loss: 0.2195 - accuracy: 0.9144 - val_loss: 0.3063 - val_accuracy: 0.8764 - 5s/epoch
Accuracy: 87.64%
```

Output 29.6: Output from evaluating the CNN model

Again, there is a lot of opportunity for further optimization, such as using deeper and/or larger convolutional layers. One interesting idea is to set the max pooling layer to use an input length of 500. This would compress each feature map to a single 32-length vector and may boost performance.

29.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Papers

Andrew L. Maas et al. "Learning Word Vectors for Sentiment Analysis". In: *The 49th Annual Meeting of the Association for Computational Linguistics*. 2011.
http://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf

APIs

IMDB movie review sentiment classification dataset. Keras API reference.
https://keras.io/api/datasets/imdb/

Embedding Layer. Keras API reference.
https://keras.io/api/layers/core_layers/embedding/

Articles

Large Movie Review Dataset.
http://ai.stanford.edu/~amaas/data/sentiment/

word embedding. Wikipedia.
https://en.wikipedia.org/wiki/Word_embedding

29.7 Summary

In this chapter, you discovered the IMDB sentiment analysis dataset for natural language processing. You learned how to develop deep learning models for sentiment analysis, including:

- How to load and review the IMDB dataset within Keras
- How to develop a large neural network model for sentiment analysis
- How to develop a one-dimensional convolutional neural network model for sentiment analysis

This is the end of this part. In the next part, you will see a different kind of neural network.

V

Recurrent Neural Networks

30 Crash Course in Recurrent Neural Networks

Another type of neural network is dominating difficult machine learning problems involving sequences of inputs: recurrent neural networks. Recurrent neural networks have connections that have loops, adding feedback and memory to the networks over time. This memory allows this type of network to learn and generalize across sequences of inputs rather than individual patterns.

A powerful type of Recurrent Neural Network called the Long Short-Term Memory Network has been shown to be particularly effective when stacked into a deep configuration, achieving state-of-the-art results on a diverse array of problems from language translation to automatic captioning of images and videos. In this chapter, you will get a crash course in recurrent neural networks for deep learning, acquiring just enough understanding to start using LSTM networks in Python with Keras. After reading this chapter, you will know:

- ▷ The limitations of Multilayer Perceptrons that are addressed by recurrent neural networks
- ▷ The problems that must be addressed to make Recurrent Neural networks useful
- ▷ The details of the Long Short-Term Memory networks used in applied deep learning

Let's get started.

Overview

This chapter is divided into five sections; they are:

- ▷ Support For Sequences in Neural Networks
- ▷ Recurrent Neural Networks
- ▷ Long Short-Term Memory Networks

30.1 Support for Sequences in Neural Networks

Some problem types are best framed by involving either a sequence as an input or an output. For example, consider a univariate time series problem, like the price of a stock over time. This dataset can be framed as a prediction problem for a classical feedforward multilayer perceptron network by defining a window's size (e.g., 5) and training the network to learn to make short-term predictions from the fixed-sized window of inputs.

This would work but is very limited. The window of inputs adds memory to the problem but is limited to just a fixed number of points and must be chosen with sufficient knowledge of the problem. A naive window would not capture the broader trends over minutes, hours, and days that might be relevant to making a prediction. From one prediction to the next, the network only knows about the specific inputs it is provided. Univariate time series prediction is important, but there are even more interesting problems that involve sequences. Consider the following taxonomy of sequence problems that require mapping an input to output (taken from Andrej Karpathy):

- ▷ **One-to-Many**: sequence output for image captioning
- ▷ **Many-to-One**: sequence input for sentiment classification
- ▷ **Many-to-Many**: sequence in and out for machine translation
- ▷ **Synchronized Many-to-Many**: synced sequences in and out for video classification

You can also see that a one-to-one example of an input to output would be an example of a classical feedforward neural network for a prediction task like image classification. Support for sequences as input is an important class of problem and one where deep learning has recently shown impressive results. State-of-the art results have been shown using recurrent neural networks — a type of network specifically designed for sequence problems.

30.2 Recurrent Neural Networks

Recurrent neural networks or RNNs are a special type of neural network designed for sequence problems. Given a standard feedforward multilayer perceptron network, a recurrent neural network can be thought of as the addition of loops to the architecture. For example, in a given layer, each neuron may pass its signal laterally (sideways) in addition to forward to the next layer. The output of the network may feed back as an input to the network with the next input vector. And so on.

The recurrent connections add state or memory to the network and allow it to learn broader abstractions from the input sequences. The field of recurrent neural networks is well established with popular methods. For the techniques to be effective on real problems, two major issues needed to be resolved for the network to be useful.

1. How to train the network with backpropagation
2. How to stop gradients vanishing or exploding during training

How to Train Recurrent Neural Networks

The staple technique for training feedforward neural networks is to backpropagate error and update the network weights. Backpropagation breaks down in a recurrent neural network because of the recurrent or loop connections. This was addressed with a modification of the backpropagation technique called Backpropagation Through Time or BPTT.

Instead of performing backpropagation on the recurrent network as stated, the structure of the network is unrolled, where copies of the neurons that have recurrent connections are created. For example, a single neuron with a connection to itself ($A \rightarrow A$) could be represented as two neurons with the same weight values ($A \rightarrow B$). This allows the cyclic graph of a recurrent neural network to be turned into an acyclic graph like a classic feedforward neural network, and backpropagation can be applied.

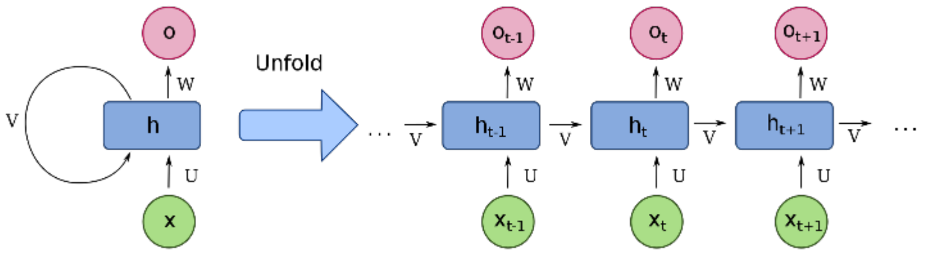


Figure 30.1: Illustration of RNN and unrolling. From Recurrent neural network *on Wikipedia*

How to Have Stable Gradients During Training

When backpropagation is used in very deep neural networks and unrolled recurrent neural networks, the gradients that are calculated to update the weights can become unstable. They can become very large numbers called exploding gradients, or very small numbers called the vanishing gradient problem. These large numbers, in turn, are used to update the weights in the network, making training unstable and the network unreliable.

This problem is alleviated in deep multilayer perceptron networks through the use of the rectifier transfer function and even more exotic but now less popular approaches of using unsupervised pre-training of layers. In recurrent neural network architectures, this problem has been alleviated using a new type of architecture called the Long Short-Term Memory Networks that allows deep recurrent networks to be trained.

30.3 Long Short-Term Memory Networks

The Long Short-Term Memory network, or LSTM network, is a recurrent neural network trained using Backpropagation Through Time and overcomes the vanishing gradient problem. As such, it can be used to create large (stacked) recurrent networks that, in turn, can be used to address difficult sequence problems in machine learning and achieve state-of-the-art results.

Instead of neurons, LSTM networks have memory blocks connected into layers. A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the block's state and output. A unit operates upon an input sequence, and each gate within a unit uses the sigmoid activation function to control whether it is triggered or not, making the change of state and addition of information flowing through the unit conditional. There are three types of gates within a memory unit:

- ▷ **Forget Gate**: conditionally decides what information to discard from the unit.
- ▷ **Input Gate**: conditionally decides which values from the input to update the memory state.
- ▷ **Output Gate**: conditionally decides what to output based on input and the memory of the unit.

Each unit is like a mini state machine where the gates of the units have weights that are learned during the training procedure. You can see how you may achieve sophisticated learning and memory from a layer of LSTMs, and it is not hard to imagine how higher-order abstractions may be layered with such multiple layers.

30.4 Further Reading

We have covered a lot of ground in this chapter. Below are some resources that you can use to go deeper into the topic of recurrent neural networks for deep learning.

Resources to learn more about recurrent neural networks and LSTMs:

Resources to learn more about Recurrent Neural Networks and LSTMs

Recurrent neural network. Wikipedia.
https://en.wikipedia.org/wiki/Recurrent_neural_network

Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. May 2015.
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Backpropagation through time. Wikipedia.
https://en.wikipedia.org/wiki/Backpropagation_through_time

Aston Zhang et al. Chapter 10, "Modern Recurrent Neural Networks". In: *Dive into Deep Learning*. 2022.
http://www.d2l.ai/chapter_recurrent-modern/deep-rnn.html

Christopher Olah. *Understanding LSTM networks*. Aug. 2015.
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long short-term memory. Wikipedia.
https://en.wikipedia.org/wiki/Long_short-term_memory

Popular tutorials for implementing LSTMs:

Recurrent Neural Networks (RNN) with Keras. TensorFlow guide.
https://www.tensorflow.org/guide/keras/rnn

Time series forecasting. TensorFlow tutorials.
https://www.tensorflow.org/tutorials/structured_data/time_series

Text generation with an RNN. TensorFlow tutorials.
https://www.tensorflow.org/text/tutorials/text_generation

Primary sources on LSTMs:

Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". *Neural Computation*, 9(8), Nov. 1997, pp. 1735–1780. DOI: 10.1162/neco.1997.9.81735.

Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". *Neural Computation*, 12(10), Oct. 2000, pp. 2451–2471. DOI: 10.1162/089976600300015015.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training Recurrent Neural Networks". *Proceedings of Machine Learning Reserach*, 28(3), 2013, pp. 1310–1318. https://arxiv.org/pdf/1211.5063v2.pdf

30.5 Summary

In this chapter, you discovered sequence problems and recurrent neural networks that can be used to address them.

Specifically, you learned:

▷ The limitations of classical feedforward neural networks and how recurrent neural networks can overcome these problems

▷ The practical problems in training recurrent neural networks and how they are overcame

▷ The Long Short-Term Memory network used to create deep recurrent neural networks

In the next chapter, you will learn some techniques on using neural network to predict time series without LSTM.

31 Time Series Prediction with Multilayer Perceptrons

Time series prediction is a difficult problem both to frame and address with machine learning. In this chapter, you will discover how to develop neural network models for time series prediction in Python using the Keras deep learning library. After reading this chapter, you will know:

- ▷ About the airline passengers univariate time series prediction problem
- ▷ How to phrase time series prediction as a regression problem and develop a neural network model for it
- ▷ How to frame time series prediction with a time lag and develop a neural network model for it

Let's get started.

Overview

This chapter is divided into five sections; they are:

- ▷ The Time Series Prediction Problem
- ▷ Multilayer Perceptron Regression
- ▷ Multilayer Perceptron Using the Window Method

31.1 The Time Series Prediction Problem

The problem you will look at in this chapter is the international airline passengers prediction problem. This is a problem where, given a year and a month, the task is to predict the number of international airline passengers in units of 1,000. The data ranges from January 1949 to December 1960, or 12 years, with 144 observations. The data is from a book by Box, Jenkins, and Reinsel. You can get it from the source code bundle of this book or download[1] the dataset in CSV format and save it with the filename `international-airline-passengers.csv`. Below is a sample of the first few lines of the file.

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv

```
"Month","Passengers"
"1949-01",112
"1949-02",118
"1949-03",132
"1949-04",129
```

Output 31.1: Sample output from evaluating the baseline model

You can load this dataset easily using the Pandas library. You are not interested in the date, given that each observation is separated by the same interval of one month. Therefore, when you load the dataset, you can exclude the first column. Once loaded, you can easily plot the whole dataset. The code to load and plot the dataset is listed below.

```python
import pandas as pd
import matplotlib.pyplot as plt
dataset = pd.read_csv('airline-passengers.csv', usecols=[1], engine='python')
plt.plot(dataset)
plt.show()
```

Listing 31.1: Load and plot the time series dataset

You can see an upward trend in the plot. You can also see some periodicity in the dataset that probably corresponds to the northern hemisphere summer holiday period.

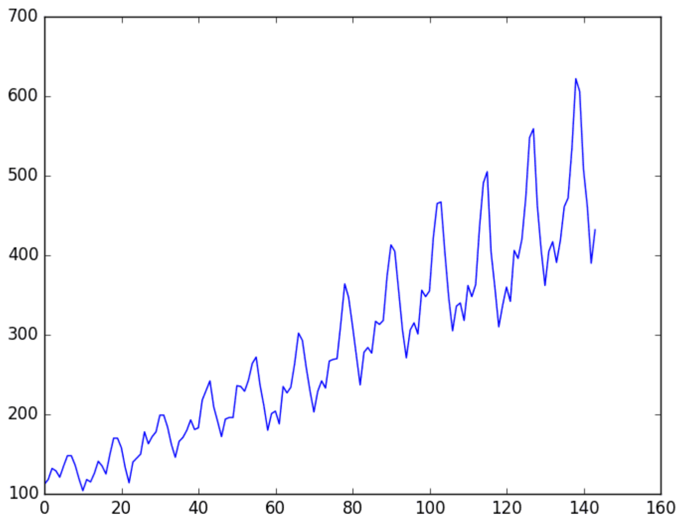


Figure 31.1: Plot of the airline passengers dataset

Let's keep things simple and work with the data as-is. Normally, it is a good idea to investigate various data preparation techniques to rescale the data and make it stationary.

31.2 Multilayer Perceptron Regression

You want to phrase the time series prediction problem as a regression problem. That is, given the number of passengers (in units of thousands) this month, what is the number of passengers next month? You can write a simple function to convert your single column of data into a two-column dataset: The first column containing this month's (t) passenger count and the second column containing next month's ($t + 1$) passenger count to be predicted. Before you get started, let's first import all the functions and classes you will need to use.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
...
```

Listing 31.2: Import classes and functions

You can also use the code from the previous section to load the dataset as a Pandas dataframe. You can then extract the NumPy array from the dataframe and convert the integer values to floating point values, which are more suitable for modeling with a neural network.

```
...
# load the dataset
dataframe = pd.read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
```

Listing 31.3: Load the time series dataset

After you model the data and estimate the skill of your model on the training dataset, you need to get an idea of the skill of the model on new unseen data. For a normal classification or regression problem, you would do this using k-fold cross-validation. With time series data, the sequence of values is important. A simple method that you can use is to split the ordered dataset into train and test datasets. The code below calculates the index of the split point and separates the data into the training datasets with 67% of the observations used to train your model, leaving the remaining 33% for testing the model.

```
...
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))
```

Listing 31.4: Split dataset into train and test

Now, you can define a function to create a new dataset as described above. The function takes two arguments: the dataset, which is a NumPy array that you want to convert into a dataset, and the `look_back`, which is the number of previous time steps to use as input variables to predict the next time period, in this case, defaulted to 1. This default will create a dataset where X is the number of passengers at a given time (t), and Y is the number of passengers at the next time ($t + 1$). It can be configured, and you will look at constructing a differently shaped dataset in the next section.

```
...
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
```

Listing 31.5: Function to prepare dataset for modeling

Let's take a look at the effect of this function on the first few rows of the dataset.

```
X         Y
112       118
118       132
132       129
129       121
121       135
```

Output 31.2: Sample of the prepared dataset

If you compare these first five rows to the original dataset sample listed in the previous section, you can see the $X = t$ and $Y = t+1$ pattern in the numbers. Let's use this function to prepare the train and test datasets for modeling.

```
...
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
```

Listing 31.6: Call function to prepare dataset for modeling

We can now fit a multilayer perceptron model to the training data. We use a simple network with one input, one hidden layer with eight neurons, and an output layer. The model is fit using mean squared error, which, if you take the square root, gives you an error score in the units of the dataset. I tried a few rough parameters and settled on the configuration below, but by no means is the network listed optimized.

```
...
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(8, input_shape=(look_back,), activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=200, batch_size=2, verbose=2)
```

Listing 31.7: Define and fit multilayer perceptron model

Once the model is fit, you can estimate the performance of the model on the train and test datasets. This will give you a point of comparison for new models.

```
...
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))
```

Listing 31.8: Evaluate the fit model

Finally, you can generate predictions using the model for both the train and test dataset to get a visual indication of the skill of the model. Because of how the dataset was prepared, you must shift the predictions to align on the x-axis with the original dataset. Once prepared, the data is plotted, showing the original dataset in blue, the predictions for the training dataset in green, and the predictions on the unseen test dataset in red.

```
...
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)

# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict

# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict

# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 31.9: Generate and plot predictions

Tying this all together, the complete example is listed below.

```
# Multilayer Perceptron to Predict International Airline Passengers (t+1, given t)
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)

# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(8, input_shape=(look_back,), activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=200, batch_size=2, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset, 'b')
plt.plot(trainPredictPlot, 'g')
plt.plot(testPredictPlot, 'r')
plt.show()
```

Listing 31.10: Multilayer perceptron model for the $t+1$ model

Running the example reports the model performance.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Taking the square root of the performance estimates, you can see that the model has an average error of 22 passengers (in thousands) on the training dataset and 46 passengers (in thousands) on the test dataset.

```
...
Epoch 395/400
46/46 - 0s - loss: 513.2617 - 13ms/epoch - 275us/step
Epoch 396/400
46/46 - 0s - loss: 494.1868 - 12ms/epoch - 268us/step
Epoch 397/400
46/46 - 0s - loss: 483.3908 - 12ms/epoch - 268us/step
Epoch 398/400
46/46 - 0s - loss: 501.8111 - 13ms/epoch - 281us/step
Epoch 399/400
46/46 - 0s - loss: 523.2578 - 13ms/epoch - 280us/step
Epoch 400/400
46/46 - 0s - loss: 513.7587 - 12ms/epoch - 263us/step
Train Score: 487.39 MSE (22.08 RMSE)
Test Score: 2070.68 MSE (45.50 RMSE)
```

Output 31.3: Sample output of the multilayer perceptron model for the $t + 1$ model

From the plot, you can see that the model did a pretty poor job of fitting both the training and the test datasets. It basically predicted the same input value as the output.

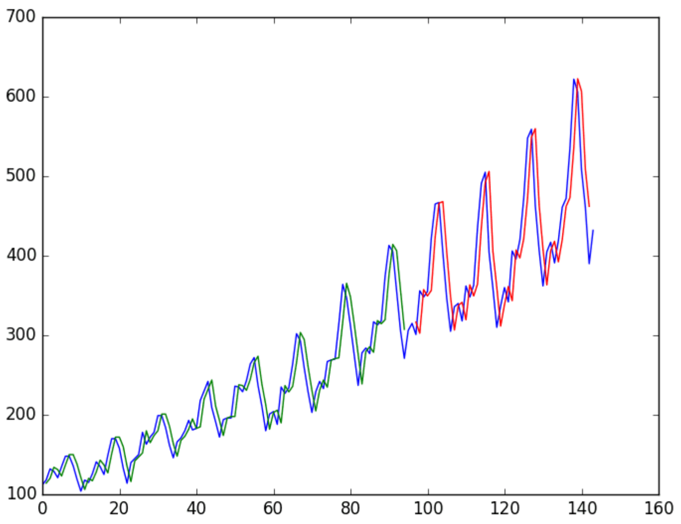


Figure 31.2: Naive time series predictions with neural network. Blue=Whole dataset; Green=Training; Red=Predictions

31.3 Multilayer Perceptron Using the Window Method

You can also phrase the problem so that multiple recent time steps can be used to make the prediction for the next time step. This is called the window method, and the size of the window is a parameter that can be tuned for each problem. For example, given the current time (t) to predict the value at the next time in the sequence ($t+1$), you can use the current time (t) as well as the two prior times ($t-1$ and $t-2$). When phrased as a regression problem, the input variables are $t-2$, $t-1$, t and the output variable is $t+1$.

The `create_dataset()` function used in the previous section allows you to create this formulation of the time series problem by increasing the `look_back` argument from 1 to 3. A sample of the dataset with this formulation looks as follows:

```
X1     X2     X3     Y
112    118    132    129
118    132    129    121
132    129    121    135
129    121    135    148
121    135    148    148
```

Output 31.4: Sample dataset of the window formulation of the problem

You can re-run the example in the previous section with the larger window size. You will increase the network capacity to handle the additional information. The first hidden layer is increased to 14 neurons, and a second hidden layer is added with eight neurons. The number of epochs is also increased to 400.

The whole code listing with just the window size change is listed below for completeness.

```
# Multilayer Perceptron to Predict International Airline Passengers (t+1, given
# t, t-1, t-2)
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)

# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape dataset
look_back = 3
```

```
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(12, input_shape=(look_back,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=400, batch_size=2, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore, math.sqrt(trainScore)))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore, math.sqrt(testScore)))
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset, 'b')
plt.plot(trainPredictPlot, 'g')
plt.plot(testPredictPlot, 'r')
plt.show()
```

Listing 31.11: Multilayer perceptron model for the window method

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example provides the following output.

```
Epoch 395/400
46/46 - 0s - loss: 419.0309 - 14ms/epoch - 294us/step
Epoch 396/400
46/46 - 0s - loss: 429.3398 - 14ms/epoch - 300us/step
Epoch 397/400
46/46 - 0s - loss: 412.2588 - 14ms/epoch - 298us/step
Epoch 398/400
46/46 - 0s - loss: 424.6126 - 13ms/epoch - 292us/step
Epoch 399/400
46/46 - 0s - loss: 429.6443 - 14ms/epoch - 296us/step
Epoch 400/400
46/46 - 0s - loss: 419.9067 - 14ms/epoch - 301us/step
```

```
Train Score: 393.07 MSE (19.83 RMSE)
Test Score: 1833.35 MSE (42.82 RMSE)
```

Output 31.5: Sample output of the multilayer perceptron model for the window model

You can see that the error was not significantly reduced compared to that of the previous section. Looking at the graph, you can see more structure in the predictions. Again, the window size and the network architecture were not tuned; this is just a demonstration of how to frame a prediction problem. Taking the square root of the performance scores, you can see the average error on the training dataset was 20 passengers (in thousands per month), and the average error on the unseen test set was 43 passengers (in thousands per month).

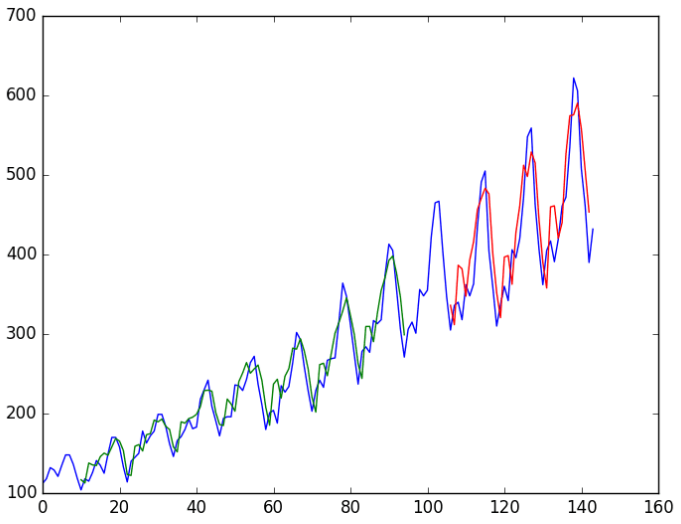


Figure 31.3: Window method for time series predictions with neural networks. Blue=Whole dataset; Green=Training; Red=Predictions

31.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Books

G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis, Forecasting and Control.* 3rd ed. Holden-Day, 1976.

31.5 Summary

In this chapter, you discovered how to develop a neural network model for a time series prediction problem using the Keras deep learning library. After working through this chapter, you now know:

- About the international airline passenger prediction time series dataset
- How to frame time series prediction problems as regression problems and develop a neural network model
- How use the window approach to frame a time series prediction problem and develop a neural network model

In the next chapter, you will see how LSTM can be used for the same problem.

32 Time Series Prediction with LSTM Networks

Time series prediction problems are a difficult type of predictive modeling problem. Unlike regression predictive modeling, time series also adds the complexity of a sequence dependence among the input variables. A powerful type of neural network designed to handle sequence dependence is called a recurrent neural network. The Long Short-Term Memory network or LSTM network is a type of recurrent neural network used in deep learning because very large architectures can be successfully trained.

In this chapter, you will discover how to develop LSTM networks in Python using the Keras deep learning library to address a demonstration time series prediction problem. After completing this chapter, you will know how to implement and develop LSTM networks for your own time series prediction problems and other more general sequence problems. You will know:

- ▷ How to develop LSTM networks for a time series prediction problem framed as regression
- ▷ How to develop LSTM networks for a time series prediction problem using a window, for both features and time steps
- ▷ How to develop and make predictions using LSTM networks that maintain state (memory) across very long sequences

In this chapter, we will develop a number of LSTMs for a standard time series prediction problem. The problem and the chosen configuration for the LSTM networks are for demonstration purposes only. They are not optimized. These examples will show exactly how you can develop your own differently structured LSTM networks for time series predictive modeling problems.

Let's get started.

Overview

This chapter is divided into six sections; they are:

- ▷ Problem Description
- ▷ LSTM Network for Regression

- ▷ LSTM for Regression Using the Window Method
- ▷ LSTM for Regression with Time Steps
- ▷ LSTM with Memory Between Batches
- ▷ Stacked LSTMs with Memory Between Batches

32.1 Problem Description

The problem you will look at in this chapter is the international airline passengers prediction problem, described in Section 31.1. This is a problem where, given a year and a month, the task is to predict the number of international airline passengers in units of 1,000. The data ranges from January 1949 to December 1960, or 12 years, with 144 observations.

We can phrase the problem as a regression problem, as was done in the previous chapter. That is, given the number of passengers (in units of thousands) this month, what is the number of passengers next month. This example will reuse the same data loading and preparation from the previous chapter, specifically the use of the `create_dataset()` function.

We will use LSTM network, described in Section 30.3, for the prediction.

32.2 LSTM Network for Regression

You can write a simple function to convert the single column of data into a two-column dataset: the first column containing this month's (t) passenger count and the second column containing next month's $(t+1)$ passenger count, to be predicted.

Before you start, let's first import all the functions and classes you will use. This assumes a working Python environment with the Keras deep learning library installed.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

Listing 32.1: Importing classes and functions

Before you do anything, it is a good idea to fix the random number seed to ensure your results are reproducible.

```
# fix random seed for reproducibility
tf.random.set_seed(7)
```

Listing 32.2: Fix the random seed for reproducibility

You can also use the code from the previous chapter to load the dataset as a Pandas dataframe. You can then extract the NumPy array from the dataframe and convert the integer values to floating point values, which are more suitable for modeling with a neural network.

```
# load the dataset
dataframe = pd.read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
```

Listing 32.3: Load dataset as a Pandas dataframe

LSTMs are sensitive to the scale of the input data, specifically when the sigmoid (default) or tanh activation functions are used. It can be a good practice to rescale the data to the range of 0-to-1, also called normalizing. You can easily normalize the dataset using the `MinMaxScaler` preprocessing class from the scikit-learn library.

```
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

Listing 32.4: Normalize the dataset with MinMaxScaler

After you model the data and estimate the skill of your model on the training dataset, you need to get an idea of the skill of the model on new unseen data. For a normal classification or regression problem, you would do this using cross-validation.

With time series data, the sequence of values is important. A simple method that you can use is to split the ordered dataset into train and test datasets. The code below calculates the index of the split point and separates the data into the training datasets, with 67% of the observations used to train the model, leaving the remaining 33% for testing the model.

```
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))
```

Listing 32.5: Split the data into training and test sets

Now, you can define a function to create a new dataset, as described above. The function takes two arguments: the `dataset`, which is a NumPy array you want to convert into a dataset, and the `look_back`, which is the number of previous time steps to use as input variables to predict the next time period — in this case, defaulted to 1. This default will create a dataset where X is the number of passengers at a given time (t), and Y is the number of passengers at the next time ($t + 1$). It can be configured by constructing a differently shaped dataset in the next section.

```
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
```

Listing 32.6: Helper function to convert a time series into a dataset matrix

Let's take a look at the effect of this function on the first rows of the dataset (shown in the unnormalized form for clarity).

```
X          Y
112        118
118        132
132        129
129        121
121        135
```

Output 32.1: Sample dataset as prepared

If you compare these first five rows to the original dataset sample listed in the previous section, you can see the $X = t$ and $Y = t + 1$ pattern in the numbers.

Let's use this function to prepare the train and test datasets for modeling.

```
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
```

Listing 32.7: Prepare the train and test datasets

The LSTM network expects the input data (X) to be provided with a specific array structure in the form of [*samples, time steps, features*]. Currently, the data is in the form of [*samples, features*], and you are framing the problem as one time step for each sample. You can transform the prepared train and test input data into the expected structure using `numpy.reshape()` as follows:

```
# reshape input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

Listing 32.8: Reshape the prepared dataset for the LSTM layers

You are now ready to design and fit your LSTM network for this problem. The network has a visible layer with 1 input, a hidden layer with 4 LSTM blocks or neurons, and an output layer that makes a single value prediction. The default sigmoid activation function is used for the LSTM memory blocks. The network is trained for 100 epochs, and a batch size of 1 is used.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

Listing 32.9: Define and fit the LSTM network

Once the model is fit, you can estimate the performance of the model on the train and test datasets. This will give you a point of comparison for new models. Note that you will invert the predictions before calculating error scores to ensure that performance is reported in the same units as the original data (thousands of passengers per month).

```
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = np.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = np.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
```

Listing 32.10: Estimate the RMSE of prediction using LSTM network

Finally, you can generate predictions using the model for both the train and test dataset to get a visual indication of the skill of the model. Because of how the dataset was prepared, you must shift the predictions so that they align on the x-axis with the original dataset. Once prepared, the data is plotted, showing the original dataset in blue, the predictions for the training dataset in green, and the predictions on the unseen test dataset in red.

```
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 32.11: Align predictions and plot

You can see that the model fit both the training and the test datasets.

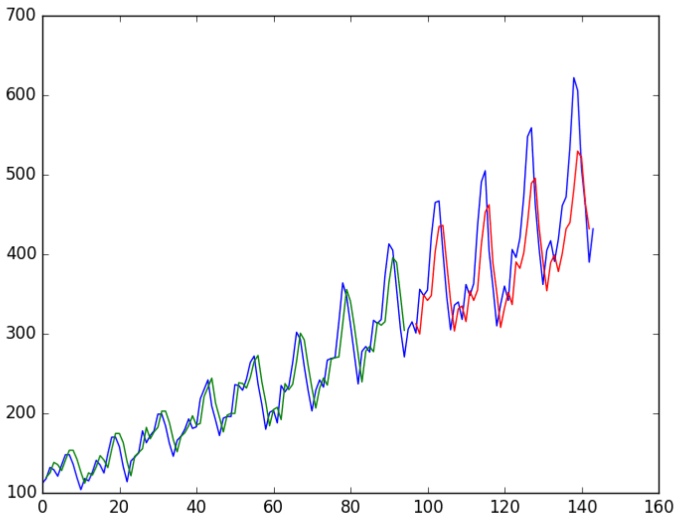


Figure 32.1: LSTM trained on regression formulation of passenger prediction problem

For completeness, below is the entire code example.

```
# LSTM for international airline passengers problem with regression framing
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import math
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
```

```
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = np.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = np.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 32.12: LSTM model on the $t + 1$ problem

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running the example produces the following output.

```
...
Epoch 95/100
94/94 - 0s - loss: 0.0021 - 37ms/epoch - 391us/step
Epoch 96/100
94/94 - 0s - loss: 0.0020 - 37ms/epoch - 398us/step
Epoch 97/100
94/94 - 0s - loss: 0.0020 - 37ms/epoch - 396us/step
Epoch 98/100
94/94 - 0s - loss: 0.0020 - 37ms/epoch - 391us/step
Epoch 99/100
94/94 - 0s - loss: 0.0020 - 37ms/epoch - 394us/step
Epoch 100/100
94/94 - 0s - loss: 0.0020 - 36ms/epoch - 382us/step
3/3 [==============================] - 0s 490us/step
2/2 [==============================] - 0s 461us/step
Train Score: 22.68 RMSE
Test Score: 49.34 RMSE
```

Output 32.2: Sample output from the LSTM model on the $t+1$ problem

You can see that the model has an average error of about 23 passengers (in thousands) on the training dataset and about 49 passengers (in thousands) on the test dataset. Not that bad.

32.3 LSTM for Regression Using the Window Method

You can also phrase the problem so that multiple, recent time steps can be used to make the prediction for the next time step. This is called a window, and the size of the window is a parameter that can be tuned for each problem. For example, given the current time (t) to predict the value at the next time in the sequence ($t+1$), you can use the current time (t), as well as the two prior times ($t-1$ and $t-2$) as input variables. When phrased as a regression problem, the input variables are $t-2$, $t-1$, and t, and the output variable is $t+1$. In this example, a window of three time steps is used. But the size of a window is a hyperparameter that you can adjust, for example, using grid search.

The `create_dataset()` function created in the previous section allows you to create this formulation of the time series problem by increasing the `look_back` argument from 1 to 3. A sample of the dataset with this formulation is as follows:

```
X1      X2      X3      Y
112     118     132     129
118     132     129     121
132     129     121     135
129     121     135     148
121     135     148     148
```

Output 32.3: Sample dataset of the window formulation of the problem

We can re-run the example in the previous section with the larger window size. The whole code listing with just the window size change is listed below for completeness.

```
# LSTM for international airline passengers problem with window regression framing
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = np.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
```

```
print('Train Score: %.2f RMSE' % (trainScore))
testScore = np.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 32.13: LSTM for the window model

Running the example provides the following output:

```
Epoch 95/100
92/92 - 0s - loss: 0.0023 - 35ms/epoch - 384us/step
Epoch 96/100
92/92 - 0s - loss: 0.0023 - 36ms/epoch - 389us/step
Epoch 97/100
92/92 - 0s - loss: 0.0024 - 37ms/epoch - 404us/step
Epoch 98/100
92/92 - 0s - loss: 0.0023 - 36ms/epoch - 392us/step
Epoch 99/100
92/92 - 0s - loss: 0.0022 - 36ms/epoch - 389us/step
Epoch 100/100
92/92 - 0s - loss: 0.0022 - 35ms/epoch - 384us/step
3/3 [==============================] - 0s 514us/step
2/2 [==============================] - 0s 533us/step
Train Score: 24.86 RMSE
Test Score: 70.48 RMSE
```

Output 32.4: Sample output from the LSTM model on the window problem

You can see that the error was increased slightly compared to that of the previous section. The window size and the network architecture were not tuned: This is just a demonstration of how to frame a prediction problem.

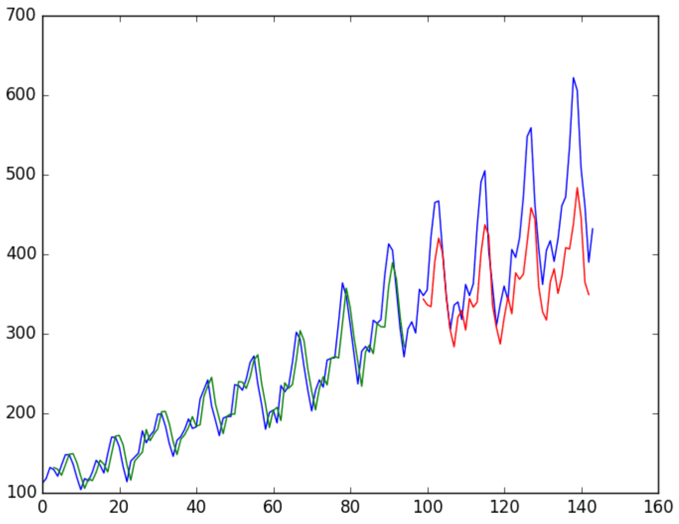


Figure 32.2: LSTM trained on window method formulation of passenger prediction problem

32.4 LSTM for Regression with Time Steps

You may have noticed that the data preparation for the LSTM network includes time steps. Some sequence problems may have a varied number of time steps per sample. For example, you may have measurements of a physical machine leading up to the point of failure or a point of surge. Each incident would be a sample of observations that lead up to the event, which would be the time steps, and the variables observed would be the features. Time steps provide another way to phrase your time series problem. Like above in the window example, you can take prior time steps in your time series as inputs to predict the output at the next time step.

Instead of phrasing the past observations as separate input features, you can use them as time steps of the one input feature, which is indeed a more accurate framing of the problem. You can do this using the same data representation as in the previous window-based example, except when you reshape the data, you set the columns to be the time steps dimension and change the features dimension back to 1. For example:

```
# reshape input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = np.reshape(testX, (testX.shape[0], testX.shape[1], 1))
```

Listing 32.14: Reshape the prepared dataset for the LSTM layers with time steps

The entire code listing is provided below for completeness.

```
# LSTM for international airline passengers problem with time step regression framing
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = np.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(look_back, 1)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = np.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
```

```
print('Train Score: %.2f RMSE' % (trainScore))
testScore = np.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 32.15: LSTM for the time steps model

Running the example provides the following output:

```
...
Epoch 95/100
92/92 - 0s - loss: 0.0023 - 45ms/epoch - 484us/step
Epoch 96/100
92/92 - 0s - loss: 0.0023 - 45ms/epoch - 486us/step
Epoch 97/100
92/92 - 0s - loss: 0.0024 - 44ms/epoch - 479us/step
Epoch 98/100
92/92 - 0s - loss: 0.0022 - 45ms/epoch - 489us/step
Epoch 99/100
92/92 - 0s - loss: 0.0022 - 45ms/epoch - 485us/step
Epoch 100/100
92/92 - 0s - loss: 0.0021 - 45ms/epoch - 490us/step
3/3 [==============================] - 0s 635us/step
2/2 [==============================] - 0s 616us/step
Train Score: 24.84 RMSE
Test Score: 60.98 RMSE
```

Output 32.5: Sample output from the LSTM model on the time step problem

You can see that the results are slightly better than the previous example. However, the structure of the input data makes a lot more sense.

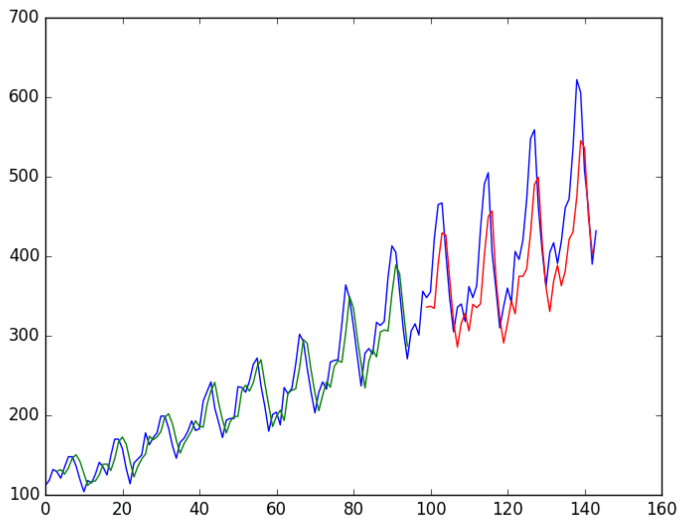


Figure 32.3: LSTM trained on time step formulation of passenger prediction problem

32.5 LSTM with Memory Between Batches

The LSTM network has memory capable of remembering across long sequences. Normally, the state within the network is reset after each training batch when fitting the model, as well as each call to `model.predict()` or `model.evaluate()`. You can gain finer control over when the internal state of the LSTM network is cleared in Keras by making the LSTM layer *stateful*. This means it can build a state over the entire training sequence and even maintain that state if needed to make predictions.

It requires that the training data not be shuffled when fitting the network. It also requires explicit resetting of the network state after each exposure to the training data (epoch) by calls to `model.reset_states()`. This means that you must create your own outer loop of epochs and within each epoch call `model.fit()` and `model.reset_states()`. For example:

```
for i in range(100):
    model.fit(trainX, trainY, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
```

Listing 32.16: Manually reset LSTM state between epochs

Finally, when the LSTM layer is constructed, the `stateful` parameter must be set to `True`. Instead of specifying the input dimensions, you must hard code the number of samples in a batch, the number of time steps in a sample, and the number of features in a time step by setting the `batch_input_shape` parameter. For example:

```
model.add(LSTM(4, batch_input_shape=(batch_size, time_steps, features), stateful=True))
```

Listing 32.17: Set the stateful parameter on the LSTM layer

This same batch size must then be used later when evaluating the model and making predictions. For example:

```
model.predict(trainX, batch_size=batch_size)
```

Listing 32.18: Set batch size when making predictions

You can adapt the previous time step example to use a stateful LSTM. The full code listing is provided below.

```
# LSTM for international airline passengers problem with memory
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = np.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
batch_size = 1
model = Sequential()
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
for i in range(100):
    model.fit(trainX, trainY, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
# make predictions
trainPredict = model.predict(trainX, batch_size=batch_size)
model.reset_states()
testPredict = model.predict(testX, batch_size=batch_size)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = np.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = np.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 32.19: LSTM for the manual management of state

Running the example provides the following output:

```
...
92/92 - 0s - loss: 0.0024 - 46ms/epoch - 502us/step
92/92 - 0s - loss: 0.0023 - 49ms/epoch - 538us/step
92/92 - 0s - loss: 0.0023 - 47ms/epoch - 514us/step
92/92 - 0s - loss: 0.0023 - 48ms/epoch - 526us/step
92/92 - 0s - loss: 0.0022 - 48ms/epoch - 517us/step
92/92 - 0s - loss: 0.0022 - 48ms/epoch - 521us/step
92/92 - 0s - loss: 0.0022 - 47ms/epoch - 512us/step
92/92 - 0s - loss: 0.0021 - 50ms/epoch - 540us/step
92/92 - 0s - loss: 0.0021 - 47ms/epoch - 512us/step
92/92 - 0s - loss: 0.0021 - 52ms/epoch - 565us/step
92/92 [==============================] - 0s 448us/step
44/44 [==============================] - 0s 383us/step
Train Score: 24.48 RMSE
Test Score: 49.55 RMSE
```

Output 32.6: Sample output from the stateful LSTM model

You do see that results are better than some, worse than others. The model may need more modules and may need to be trained for more epochs to internalize the structure of the problem.

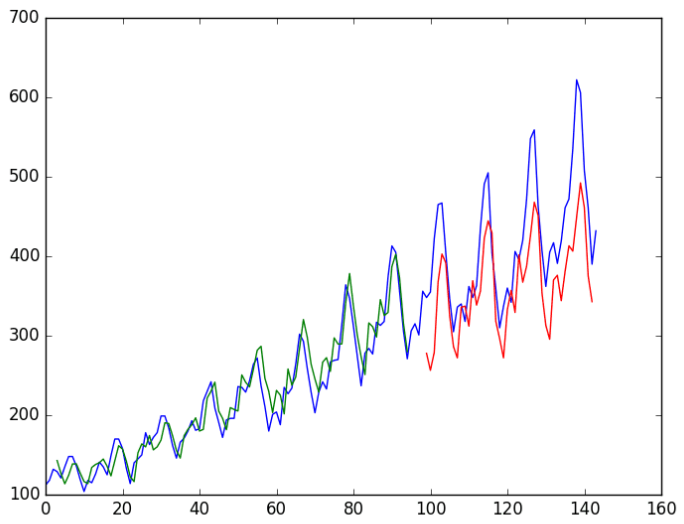


Figure 32.4: Stateful LSTM trained on regression formulation of passenger prediction problem

32.6 Stacked LSTMs with Memory Between Batches

Finally, let's take a look at one of the big benefits of LSTMs, the fact that they can be successfully trained when stacked into deep network architectures. LSTM networks can be stacked in Keras in the same way that other layer types can be stacked. One addition to the configuration that is required is that an LSTM layer prior to each subsequent LSTM layer must return the sequence. This can be done by setting the `return_sequences` parameter on the layer to `True`. You can extend the stateful LSTM in the previous section to have two layers, as follows:

```
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True,
               return_sequences=True))
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
```

Listing 32.20: Define a stacked LSTM model

The entire code listing is provided below for completeness.

```
# Stacked LSTM for international airline passengers problem with memory
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])
    return np.array(dataX), np.array(dataY)
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset
dataframe = read_csv('airline-passengers.csv', usecols=[1], engine='python')
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = np.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
batch_size = 1
model = Sequential()
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True,
               return_sequences=True))
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
for i in range(100):
    model.fit(trainX, trainY, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
# make predictions
trainPredict = model.predict(trainX, batch_size=batch_size)
model.reset_states()
testPredict = model.predict(testX, batch_size=batch_size)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = np.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
```

```
print('Train Score: %.2f RMSE' % (trainScore))
testScore = np.sqrt(mean_squared_error(testY[0], testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
# shift train predictions for plotting
trainPredictPlot = np.empty_like(dataset)
trainPredictPlot[:, :] = np.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(dataset)
testPredictPlot[:, :] = np.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 32.21: Stacked LSTM model

Running the example produces the following output.

```
...
92/92 - 0s - loss: 0.0016 - 78ms/epoch - 849us/step
92/92 - 0s - loss: 0.0015 - 80ms/epoch - 874us/step
92/92 - 0s - loss: 0.0015 - 78ms/epoch - 843us/step
92/92 - 0s - loss: 0.0015 - 78ms/epoch - 845us/step
92/92 - 0s - loss: 0.0015 - 79ms/epoch - 859us/step
92/92 - 0s - loss: 0.0015 - 78ms/epoch - 848us/step
92/92 - 0s - loss: 0.0015 - 78ms/epoch - 844us/step
92/92 - 0s - loss: 0.0015 - 78ms/epoch - 852us/step
92/92 [==============================] - 0s 563us/step
44/44 [==============================] - 0s 453us/step
Train Score: 20.58 RMSE
Test Score: 55.99 RMSE
```

Output 32.7: Sample output from the stacked LSTM model

The predictions on the test dataset are again worse. This is more evidence to suggest the need for additional training epochs.

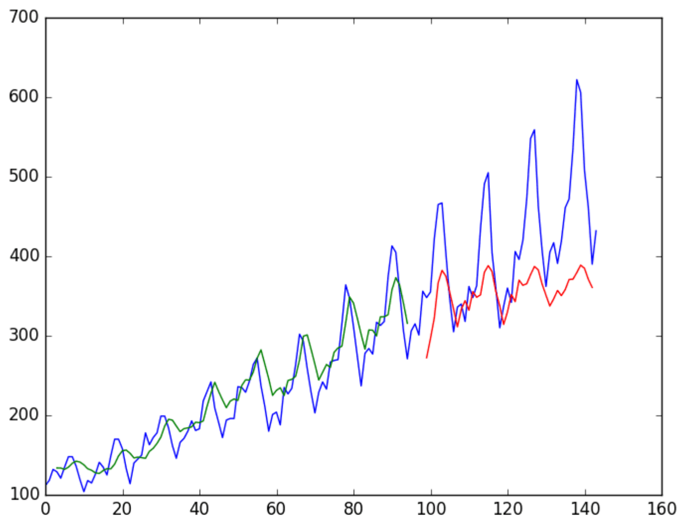


Figure 32.5: Stacked stateful LSTMs trained on regression formulation of passenger prediction problem

32.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Articles

LSTM layer. Keras API reference.
    https://keras.io/api/layers/recurrent_layers/lstm/

Time series forecasting. TensorFlow tutorials.
    https://www.tensorflow.org/tutorials/structured_data/time_series

32.8 Summary

In this chapter, you discovered how to develop LSTM recurrent neural networks for time series prediction in Python with the Keras deep learning network. Specifically, you learned:

- ▷ About the international airline passenger time series prediction problem
- ▷ How to create an LSTM for a regression and a window formulation of the time series problem
- ▷ How to create an LSTM with a time step formulation of the time series problem
- ▷ How to create an LSTM with state and stacked LSTMs with state to learn long sequences

In the next chapter, you will learn that LSTM is not only for time series problems, but also can help understanding a paragraph.

33 Project: Sequence Classification of Movie Reviews

Sequence classification is a predictive modeling problem where you have some sequence of inputs over space or time, and the task is to predict a category for the sequence. This problem is difficult because the sequences can vary in length, comprise a very large vocabulary of input symbols, and may require the model to learn the long term context or dependencies between symbols in the input sequence. In this chapter, you will discover how you can develop LSTM recurrent neural network models for sequence classification problems in Python using the Keras deep learning library. After completing this chapter, you will know:

▷ How to develop an LSTM model for a sequence classification problem

▷ How to reduce overfitting in your LSTM models through the use of dropout

▷ How to combine LSTM models with Convolutional Neural Networks that excel at learning spatial relationships

Let's get started.

Overview

This chapter is divided into five sections; they are:

▷ Problem Description

▷ Simple LSTM for Sequence Classification

▷ LSTM for Sequence Classification with Dropout

▷ Bidirectional LSTM for Sequence Classification

▷ LSTM and CNN for Sequence Classification

33.1 Problem Description

The problem that you will use to demonstrate sequence learning in this chapter is the IMDB movie review sentiment classification problem, introduced in Section 29.1.

Keras provides built-in access to the IMDB dataset. The `imdb.load_data()` function allows you to load the dataset in a format ready for use in neural networks and deep learning models. The words have been replaced by integers that indicate the ordered frequency of each word in the dataset. The sentences in each review are therefore comprised of a sequence of integers.

Word Embedding

You will map each movie review into a real vector domain, a popular technique when working with text — called word embedding. This is a technique where words are encoded as real-valued vectors in a high dimensional space, where the similarity between words in terms of meaning translates to closeness in the vector space.

Keras provides a convenient way to convert positive integer representations of words into a word embedding by an `Embedding` layer. You will map each word onto a 32-length real valued vector. You will also limit the total number of words that you are interested in modeling to the 5000 most frequent words and zero out the rest. Finally, the sequence length (number of words) in each review varies, so you will constrain each review to be 500 words, truncating long reviews and padding the shorter reviews with zero values.

Now that you have defined your problem and how the data will be prepared and modeled, you are ready to develop an LSTM model to classify the sentiment of movie reviews.

33.2 Simple LSTM for Sequence Classification

You can quickly develop a small LSTM for the IMDB problem and achieve good accuracy. Let's start by importing the classes and functions required for this model and initializing the random number generator to a constant value to ensure you can easily reproduce the results.

```python
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
# fix random seed for reproducibility
tf.random.set_seed(7)
```

Listing 33.1: Import classes and functions

You need to load the IMDB dataset. You are constraining the dataset to the top 5,000 words. You will also split the dataset into train (50%) and test (50%) sets.

```python
...
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
```

Listing 33.2: Load and split the dataset

Next, you need to truncate and pad the input sequences, so they are all the same length for modeling. The model will learn that the zero values carry no information. The sequences are not the same length in terms of content, but same-length vectors are required to perform the computation in Keras.

```
...
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
```

Listing 33.3: Left-pad sequences to all be the same length

You can now define, compile and fit your LSTM model. The first layer is the Embedded layer that uses 32-length vectors to represent each word. The next layer is the LSTM layer with 100 memory units (smart neurons). Finally, because this is a classification problem, you will use a `Dense` output layer with a single neuron and a sigmoid activation function to make 0 or 1 predictions for the two classes (good and bad) in the problem. Because it is a binary classification problem, log loss is used as the loss function (`binary_crossentropy` in Keras). The efficient ADAM optimization algorithm is used. The model is fit for only three epochs because it quickly overfits the problem. A large batch size of 64 reviews is used to space out weight updates.

```
...
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3, batch_size=64)
```

Listing 33.4: Define and fit LSTM model for the IMDB dataset

Once fit, you can estimate the performance of the model on unseen reviews.

```
...
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 33.5: Evaluate the fit model

For completeness, here is the full code listing for this LSTM network on the IMDB dataset.

```
# LSTM for sequence classification in the IMDB dataset
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 33.6: Simple LSTM model for the IMDB dataset

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running this example produces the following output.

```
Epoch 1/3
391/391 [==============================] - 124s 316ms/step - loss: 0.4525 - accuracy: 0.7794
Epoch 2/3
391/391 [==============================] - 124s 318ms/step - loss: 0.3117 - accuracy: 0.8706
Epoch 3/3
391/391 [==============================] - 126s 323ms/step - loss: 0.2526 - accuracy: 0.9003
Accuracy: 86.83%
```

Output 33.1: Output from running the simple LSTM model

You can see that this simple LSTM with little tuning achieves near state-of-the-art results on the IMDB problem. Importantly, this is a template that you can use to apply LSTM networks to your own sequence classification problems. Now, let's look at some extensions of this simple model that you may also want to bring to your own problems.

33.3 LSTM for Sequence Classification with Dropout

Recurrent neural networks like LSTM generally have the problem of overfitting. Dropout can be applied between layers using the `Dropout` Keras layer. You can do this easily by adding new

Dropout layers between the Embedding and LSTM layers and the LSTM and Dense output layers. For example:

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

Listing 33.7: Define LSTM model with dropout layers

The full code listing example above with the addition of Dropout layers is as follows:

```
# LSTM with Dropout for sequence classification in the IMDB dataset
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 33.8: LSTM model with dropout for the IMDB dataset

Running this example provides the following output.

```
Epoch 1/3
391/391 [==============================] - 117s 297ms/step - loss: 0.4721 - accuracy: 0.7664
Epoch 2/3
391/391 [==============================] - 125s 319ms/step - loss: 0.2840 - accuracy: 0.8864
Epoch 3/3
391/391 [==============================] - 135s 346ms/step - loss: 0.3022 - accuracy: 0.8772
Accuracy: 85.66%
```

Output 33.2: Output from running the LSTM model with dropout layer

You can see dropout having the desired impact on training with a slightly slower trend in convergence and, in this case, a lower final accuracy. The model could probably use a few more epochs of training and may achieve a higher skill (try it and see). Alternately, dropout can be applied to the input and recurrent connections of the memory units with the LSTM precisely and separately. Keras provides this capability with parameters on the `LSTM` layer, the `dropout`, for configuring the input dropout and `recurrent_dropout` for configuring the recurrent dropout. For example, you can modify the first example to add dropout to the input and recurrent connections as follows:

```python
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
```

Listing 33.9: Define LSTM model with dropout on gates

The full code listing with more precise LSTM dropout is listed below for completeness.

```python
# LSTM with dropout for sequence classification in the IMDB dataset
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 33.10: LSTM model with dropout on gates for the IMDB dataset

Running this example provides the following output.

```
Epoch 1/3
391/391 [==============================] - 220s 560ms/step - loss: 0.4605 - accuracy: 0.7784
Epoch 2/3
391/391 [==============================] - 219s 560ms/step - loss: 0.3158 - accuracy: 0.8773
Epoch 3/3
391/391 [==============================] - 219s 559ms/step - loss: 0.2734 - accuracy: 0.8930
Accuracy: 86.78%
```

Output 33.3: Output from running the LSTM model with dropout on the gates

You can see that the LSTM-specific dropout has a more pronounced effect on the convergence of the network than the layer-wise dropout. Like above, the number of epochs was kept constant and could be increased to see if the skill of the model could be further lifted. Dropout is a powerful technique for combating overfitting in your LSTM models, and it is a good idea to try both methods. Still, you may get better results with the gate-specific dropout provided in Keras.

33.4 Bidirectional LSTM for Sequence Classification

Sometimes, a sequence is better to be used in reversed order. In those case, you can simply reverse a vector `x` using the Python syntax `x[::-1]` before using is to train our LSTM network. Sometimes, neither the forward nor the reversed order works perfectly, but combining them will give better results. In this case, you will need a *bidirectional LSTM network*.

A bidirectional LSTM network is simply two separately LSTM networks; one feeds with forward sequence and another with reversed sequence. Then the output of the two LSTM networks is concatenated together before fed to the subsequent layers of the network. In Keras, we have the function `Bidirectional()` to clone a LSTM layer for forward-backward input and concatenate their output. For example,

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Bidirectional(LSTM(100, dropout=0.2, recurrent_dropout=0.2)))
model.add(Dense(1, activation='sigmoid'))
```

Listing 33.11: Creating a bidirectional LSTM network

Since we created not one, but two LSTM with 100 units each, this network will take twice the amount of time to train. Depending on the problem, this additional cost may be justified.

The full code listing with adding the bidirectional LSTM to the last example is listed below for completeness.

```
# LSTM with dropout for sequence classification in the IMDB dataset
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Bidirectional
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Bidirectional(LSTM(100, dropout=0.2, recurrent_dropout=0.2)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 33.12: Bidirectional LSTM model for the IMDB dataset

Running this example provides the following output.

```
Epoch 1/3
391/391 [==============================] - 405s 1s/step - loss: 0.4960 - accuracy: 0.7532
Epoch 2/3
391/391 [==============================] - 439s 1s/step - loss: 0.3075 - accuracy: 0.8744
Epoch 3/3
391/391 [==============================] - 430s 1s/step - loss: 0.2551 - accuracy: 0.9014
Accuracy: 87.69%
```

Output 33.4: Output from running the bidirectional LSTM model

It seems we can only get a slight improvement but with a significantly longer training time.

33.5 LSTM and CNN for Sequence Classification

Convolutional neural networks excel at learning the spatial structure in input data. The IMDB review data is a collection of sentences. It does have a one-dimensional spatial structure of a sequence of words in reviews, and the CNN may be able to pick out invariant features for the good and bad sentiment. This learned spatial feature may then be learned as sequences by an LSTM layer. You can easily add a one-dimensional CNN and max pooling layers after the `Embedding` layer, which then feeds the consolidated features to the LSTM. You can use a smallish set of 32 features with a small filter length of 3. The pooling layer can use the standard length of 2 to halve the feature map size.

For example, you would create the model as follows:

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Conv1D(32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
```

Listing 33.13: Define LSTM and CNN model

The full code listing with CNN and LSTM layers is listed below for completeness.

```
# LSTM and CNN for sequence classification in the IMDB dataset
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import MaxPooling1D
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
# fix random seed for reproducibility
tf.random.set_seed(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Conv1D(32, 3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 33.14: LSTM and CNN model for the IMDB dataset

Running this example provides the following output.

```
Epoch 1/3
391/391 [==============================] - 65s 163ms/step - loss: 0.4213 - accuracy: 0.7950
Epoch 2/3
391/391 [==============================] - 66s 168ms/step - loss: 0.2490 - accuracy: 0.9026
Epoch 3/3
391/391 [==============================] - 73s 188ms/step - loss: 0.1979 - accuracy: 0.9261
Accuracy: 88.45%
```

Output 33.5: Output from running the LSTM and CNN model

You can see that you achieve slightly better results than the first example, although with fewer weights and faster training time. You might expect that even better results could be achieved if this example was further extended to use dropout.

33.6 Further Reading

Below are some resources if you are interested in diving deeper into sequence prediction or this specific example.

APIs

IMDB movie review sentiment classification dataset. Keras API reference.
https://keras.io/api/datasets/imdb/

LSTM layer. Keras API reference.
https://keras.io/api/layers/recurrent_layers/lstm/

Embedding Layer. Keras API reference.
https://keras.io/api/layers/core_layers/embedding/

Articles

Large Movie Review Dataset.
http://ai.stanford.edu/~amaas/data/sentiment/

François Chollet. *Bidirectional LSTM on IMDB.* Keras code examples, 2020.
https://keras.io/examples/nlp/bidirectional_lstm_imdb/

Books

Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks.* Springer, 2012.
https://www.amazon.com/dp/3642247962/
(Preprint available online: https://www.cs.toronto.edu/~graves/preprint.pdf).

33.7 Summary

In this chapter, you discovered how to develop LSTM network models for sequence classification predictive modeling problems. Specifically, you learned:

▷ How to develop a simple single-layer LSTM model for the IMDB movie review sentiment classification problem

- ▷ How to extend your LSTM model with layer-wise and LSTM-specific dropout to reduce overfitting
- ▷ How to combine the spatial structure learning properties of a Convolutional Neural Network with the sequence learning of an LSTM

In the next chapter, you will look deeper into the states held by the LSTM network.

Understanding Stateful LSTM Recurrent Neural Networks

A powerful and popular recurrent neural network is the long short-term model network or LSTM. It is widely used because the architecture overcomes the vanishing and exploding gradient problem that plagues all recurrent neural networks, allowing very large and very deep networks to be created. Like other recurrent neural networks, LSTM networks maintain state, and the specifics of how this is implemented in a Keras framework can be confusing. In this chapter, you will discover exactly how state is maintained in LSTM networks by the Keras deep learning library. After reading this chapter, you will know:

▷ How to develop a naive LSTM network for a sequence prediction problem

▷ How to carefully manage state through batches and features with an LSTM network

▷ How to manually manage state in an LSTM network for stateful prediction

Let's get started.

Overview

This chapter is divided into seven sections; they are:

▷ Problem Description: Learn the Alphabet

▷ Naive LSTM for Learning One-Char to One-Char Mapping

▷ Naive LSTM for a Three-Char Feature Window to One-Char Mapping

▷ Naive LSTM for a Three-Char Time Step Window to One-Char Mapping

▷ LSTM State within a Batch

▷ Stateful LSTM for a One-Char to One-Char Mapping

▷ LSTM with Variable-Length Input to One-Char Output

34.1 Problem Description: Learn the Alphabet

In this chapter, you will develop and contrast a number of different LSTM recurrent neural network models. The context of these comparisons will be a simple sequence prediction problem of learning the alphabet. That is, given a letter of the alphabet, it will predict the next letter of the alphabet. This is a simple sequence prediction problem that, once understood, can be generalized to other sequence prediction problems like time series prediction and sequence classification. Let's prepare the problem with some Python code you can reuse from example to example.

First, let's import all of the classes and functions you will use in this chapter.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import to_categorical
```

Listing 34.1: Import classes and functions

Next, you can seed the random number generator to ensure that the results are the same each time the code is executed.

```
# fix random seed for reproducibility
tf.random.set_seed(7)
```

Listing 34.2: Fix the random seed for reproducibility

You can now define your dataset, the alphabet. You define the alphabet in uppercase characters for readability. Neural networks model numbers, so you need to map the letters of the alphabet to integer values. You can do this easily by creating a dictionary (map) of the letter index to the character. You can also create a reverse lookup for converting predictions back into characters to be used later.

```
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
```

Listing 34.3: Define the alphabet dataset

Now, you need to create your input and output pairs on which to train your neural network. You can do this by defining an input sequence length, then reading sequences from the input alphabet sequence. For example, use an input length of 1. Starting at the beginning of the raw input data, you can read off the first letter "`A`" and the next letter as the prediction "`B`." You move along one character and repeat until we reach a prediction of "`Z`."

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print(seq_in, '->', seq_out)
```

Listing 34.4: Create patterns from dataset

Also, print out the input pairs for sanity checking. Running the code to this point will produce the following output, summarizing input sequences of length 1 and a single output character.

```
A -> B
B -> C
C -> D
D -> E
E -> F
F -> G
G -> H
H -> I
I -> J
J -> K
K -> L
L -> M
M -> N
N -> O
O -> P
P -> Q
Q -> R
R -> S
S -> T
T -> U
U -> V
V -> W
W -> X
X -> Y
Y -> Z
```

Output 34.1: Sample alphabet training patterns

You need to reshape the NumPy array into a format expected by the LSTM networks, specifically [*samples, time steps, features*].

```
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (len(dataX), seq_length, 1))
```

Listing 34.5: Reshape training patterns for LSTM layers

Once reshaped, you can then normalize the input integers to the range 0-to-1, the range of the sigmoid activation functions used by the LSTM network.

```
# normalize
X = X / float(len(alphabet))
```

Listing 34.6: Normalize training patterns

Finally, you can think of this problem as a sequence classification task, where each of the 26 letters represents a different class. As such, you can convert the output (y) to a one-hot encoding using the Keras built-in function `to_categorical()`.

```
# one-hot encode the output variable
y = to_categorical(dataY)
```

Listing 34.7: One-hot encode output patterns

You are now ready to fit different LSTM models.

34.2 Naive LSTM for Learning One-Char to One-Char Mapping

Let's start by designing a simple LSTM to learn how to predict the next character in the alphabet, given the context of just one character. You will frame the problem as a random collection of one-letter input to one-letter output pairs. As you will see, this is a problematic framing of the problem for the LSTM to learn. Let's define an LSTM network with 32 units and an output layer using the softmax activation function for making predictions. Because this is a multiclass classification problem, you can use the log loss function (called "`categorical_crossentropy`" in Keras) and optimize the network using the ADAM optimization function. The model is fit over 500 epochs with a batch size of 1.

```
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
```

Listing 34.8: Define and fit LSTM network model

After you fit the model, you can evaluate and summarize the performance of the entire training dataset.

```
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 34.9: Evaluate the fit LSTM network model

You can then re-run the training data through the network and generate predictions, converting both the input and output pairs back into their original character format to get a visual idea of how well the network learned the problem.

```
# demonstrate some model predictions
for pattern in dataX:
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print(seq_in, "->", result)
```

Listing 34.10: Make predictions using the fit LSTM network

The entire code listing is provided below for completeness.

```
# Naive LSTM to learn one-char to one-char mapping
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import to_categorical
# fix random seed for reproducibility
tf.random.set_seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print(seq_in, '->', seq_out)
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (len(dataX), seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one-hot encode the output variable
y = to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print(seq_in, "->", result)
```

Listing 34.11: LSTM network for one-char to one-char mapping

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

Running this example produces the following output.

```
Model Accuracy: 84.00%
['A'] -> B
['B'] -> C
['C'] -> D
['D'] -> E
['E'] -> F
['F'] -> G
['G'] -> H
['H'] -> I
['I'] -> J
['J'] -> K
['K'] -> L
['L'] -> M
['M'] -> N
['N'] -> O
['O'] -> P
['P'] -> Q
['Q'] -> R
['R'] -> S
['S'] -> T
['T'] -> U
['U'] -> W
['V'] -> Y
['W'] -> Z
['X'] -> Z
['Y'] -> Z
```

Output 34.2: Output from the one-char to one-char mapping

You can see that this problem is indeed difficult for the network to learn. The reason is that the poor LSTM units do not have any context to work with. Each input-output pattern is shown to the network in a random order, and the state of the network is reset after each pattern (each batch where each batch contains one pattern). This is an abuse of the LSTM network architecture, treating it like a standard multilayer perceptron. Next, let's try a different framing of the problem to provide more sequence to the network from which to learn.

34.3 Naive LSTM for a Three-Char Feature Window to One-Char Mapping

A popular approach to adding more context to data for multilayer perceptrons is to use the window method. This is where previous steps in the sequence are provided as additional input features to the network. You can try the same trick to provide more context to the LSTM network. Here, you will increase the sequence length from 1 to 3, for example:

```
...
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
```

Listing 34.12: Increase sequence length

This creates training patterns like this:

```
ABC -> D
BCD -> E
CDE -> F
```

Output 34.3: Sample of longer input sequence length

Each element in the sequence is then provided as a new input feature to the network. This requires a modification of how the input sequences are reshaped in the data preparation step:

```
...
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (len(dataX), 1, seq_length))
```

Listing 34.13: Reshape input so sequence is features

It also requires modifying how the sample patterns are reshaped when demonstrating predictions from the model.

```
...
x = np.reshape(pattern, (1, 1, len(pattern)))
```

Listing 34.14: Reshape input for predictions so sequence is features

The entire code listing is provided below for completeness.

```
# Naive LSTM to learn three-char window to one-char mapping
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import to_categorical
# fix random seed for reproducibility
tf.random.set_seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print(seq_in, '->', seq_out)
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (len(dataX), 1, seq_length))
# normalize
X = X / float(len(alphabet))
# one-hot encode the output variable
y = to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
    x = np.reshape(pattern, (1, 1, len(pattern)))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print(seq_in, "->", result)
```

Listing 34.15: LSTM network for three-char features to one-char mapping

Running this example provides the following output.

```
Model Accuracy: 86.96%
['A', 'B', 'C'] -> D
['B', 'C', 'D'] -> E
['C', 'D', 'E'] -> F
['D', 'E', 'F'] -> G
['E', 'F', 'G'] -> H
['F', 'G', 'H'] -> I
['G', 'H', 'I'] -> J
['H', 'I', 'J'] -> K
['I', 'J', 'K'] -> L
['J', 'K', 'L'] -> M
['K', 'L', 'M'] -> N
```

```
['L', 'M', 'N'] -> O
['M', 'N', 'O'] -> P
['N', 'O', 'P'] -> Q
['O', 'P', 'Q'] -> R
['P', 'Q', 'R'] -> S
['Q', 'R', 'S'] -> T
['R', 'S', 'T'] -> U
['S', 'T', 'U'] -> V
['T', 'U', 'V'] -> Y
['U', 'V', 'W'] -> Z
['V', 'W', 'X'] -> Z
['W', 'X', 'Y'] -> Z
```

Output 34.4: Output from the three-char features to one-char mapping

You can see a slight lift in performance that may or may not be real. This is a simple problem that you were still not able to learn with LSTMs even with the window method. Again, this is a misuse of the LSTM network by a poor framing of the problem. Indeed, the sequences of letters are time steps of one feature rather than one time step of separate features. You have given more context to the network but not more sequence as expected. In the next section, you will give more context to the network in the form of time steps.

34.4 Naive LSTM for a Three-Char Time Step Window to One-Char Mapping

In Keras, the intended use of LSTMs is to provide context in the form of time steps, rather than windowed features like with other network types. You can take your first example and simply change the sequence length from 1 to 3.

```
seq_length = 3
```

Listing 34.16: Increase sequence length

Again, this creates input-output pairs that look like this:

```
ABC -> D
BCD -> E
CDE -> F
DEF -> G
```

Output 34.5: Sample of longer input sequence length

The difference is that the reshaping of the input data takes the sequence as a time step sequence of one feature rather than a single time step of multiple features.

```
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (len(dataX), seq_length, 1))
```

Listing 34.17: Reshape input so sequence is time steps

This is the correct intended use of providing sequence context to your LSTM in Keras. The full code example is provided below for completeness.

```python
# Naive LSTM to learn three-char time steps to one-char mapping
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import to_categorical
# fix random seed for reproducibility
tf.random.set_seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print(seq_in, '->', seq_out)
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (len(dataX), seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one-hot encode the output variable
y = to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print(seq_in, "->", result)
```

Listing 34.18: LSTM network for three-char time steps to one-char mapping

Running this example provides the following output.

```
Model Accuracy: 100.00%
['A', 'B', 'C'] -> D
['B', 'C', 'D'] -> E
['C', 'D', 'E'] -> F
['D', 'E', 'F'] -> G
['E', 'F', 'G'] -> H
['F', 'G', 'H'] -> I
['G', 'H', 'I'] -> J
['H', 'I', 'J'] -> K
['I', 'J', 'K'] -> L
['J', 'K', 'L'] -> M
['K', 'L', 'M'] -> N
['L', 'M', 'N'] -> O
['M', 'N', 'O'] -> P
['N', 'O', 'P'] -> Q
['O', 'P', 'Q'] -> R
['P', 'Q', 'R'] -> S
['Q', 'R', 'S'] -> T
['R', 'S', 'T'] -> U
['S', 'T', 'U'] -> V
['T', 'U', 'V'] -> W
['U', 'V', 'W'] -> X
['V', 'W', 'X'] -> Y
['W', 'X', 'Y'] -> Z
```

Output 34.6: Output from the three-char time steps to one-char mapping

You can see that the model learns the problem perfectly as evidenced by the model evaluation and the example predictions. But it has learned a simpler problem. Specifically, it has learned to predict the next letter from a sequence of three letters in the alphabet. It can be shown any random sequence of three letters from the alphabet and predict the next letter.

It cannot actually enumerate the alphabet. It's possible a large enough multilayer perceptron network might be able to learn the same mapping using the window method. The LSTM networks are stateful. They should be able to learn the whole alphabet sequence, but by default, the Keras implementation resets the network state after each training batch.

34.5 LSTM State within a Batch

The Keras implementation of LSTMs resets the state of the network after each batch. This suggests that if you had a batch size large enough to hold all input patterns and if all the input patterns were ordered sequentially, the LSTM could use the context of the sequence within the batch to better learn the sequence. You can demonstrate this easily by modifying the first example for learning a one-to-one mapping and increasing the batch size from 1 to the size of the training dataset. Additionally, Keras shuffles the training dataset before each training epoch. To ensure the training data patterns remain sequential, you can disable this shuffling.

```
...
model.fit(X, y, epochs=5000, batch_size=len(dataX), verbose=2, shuffle=False)
```

Listing 34.19: Increase base size to cover entire dataset

The network will learn the mapping of characters using within-batch sequence, but this context will not be available to the network when making predictions. You can evaluate both the ability of the network to make predictions randomly and in sequence. The full code example is provided below for completeness.

```
# Naive LSTM to learn one-char to one-char mapping with all data in each batch
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.sequence import pad_sequences
# fix random seed for reproducibility
tf.random.set_seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print(seq_in, '->', seq_out)
# convert list of lists to array and pad sequences if needed
X = pad_sequences(dataX, maxlen=seq_length, dtype='float32')
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (X.shape[0], seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one-hot encode the output variable
y = to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(16, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=5000, batch_size=len(dataX), verbose=2, shuffle=False)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
```

```
for pattern in dataX:
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print(seq_in, "->", result)
# demonstrate predicting random patterns
print("Test a Random Pattern:")
for i in range(0,20):
    pattern_index = np.random.randint(len(dataX))
    pattern = dataX[pattern_index]
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print(seq_in, "->", result)
```

Listing 34.20: LSTM network for one-char to one-char mapping within batch

Running the example provides the following output.

```
Model Accuracy: 100.00%
['A'] -> B
['B'] -> C
['C'] -> D
['D'] -> E
['E'] -> F
['F'] -> G
['G'] -> H
['H'] -> I
['I'] -> J
['J'] -> K
['K'] -> L
['L'] -> M
['M'] -> N
['N'] -> O
['O'] -> P
['P'] -> Q
['Q'] -> R
['R'] -> S
['S'] -> T
['T'] -> U
['U'] -> V
['V'] -> W
['W'] -> X
['X'] -> Y
['Y'] -> Z
Test a Random Pattern:
['T'] -> U
['V'] -> W
```

```
['M'] -> N
['Q'] -> R
['D'] -> E
['V'] -> W
['T'] -> U
['U'] -> V
['J'] -> K
['F'] -> G
['N'] -> O
['B'] -> C
['M'] -> N
['F'] -> G
['F'] -> G
['P'] -> Q
['A'] -> B
['K'] -> L
['W'] -> X
['E'] -> F
```

Output 34.7: Output from the one-char to one-char mapping within batch

As expected, the network is able to use the within-sequence context to learn the alphabet, achieving 100% accuracy on the training data. Importantly, the network can make accurate predictions for the next letter in the alphabet for randomly selected characters. Very impressive.

34.6 Stateful LSTM for a One-Char to One-Char Mapping

You have seen that you can break up the raw data into fixed-size sequences and that this representation can be learned by the LSTM but only to learn random mappings of 3 characters to 1 character. You have also seen that you can pervert the batch size to offer more sequence to the network, but only during training. Ideally, you want to expose the network to the entire sequence and let it learn the inter-dependencies rather than you defining those dependencies explicitly in the framing of the problem.

You can do this in Keras by making the LSTM layers stateful and manually resetting the state of the network at the end of the epoch, which is also the end of the training sequence. This is truly how the LSTM networks are intended to be used. You first need to define your LSTM layer as stateful. In so doing, you must explicitly specify the batch size as a dimension on the input shape. This also means that when you evaluate the network or make predictions, you must also specify and adhere to this same batch size. This is not a problem now as you are using a batch size of 1. This could introduce difficulties when making predictions when the batch size is not one, as predictions will need to be made in the batch and the sequence.

```
...
batch_size = 1
model.add(LSTM(50, batch_input_shape=(batch_size, X.shape[1], X.shape[2]), stateful=True))
```

Listing 34.21: Define a stateful LSTM layer

An important difference in training the stateful LSTM is that you manually train it one epoch at a time and reset the state after each epoch. You can do this in a `for` loop. Again, do not shuffle the input, preserving the sequence in which the input training data was created.

```
for i in range(300):
    model.fit(X, y, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
```

Listing 34.22: Manually manage LSTM state for each epoch

As mentioned, you specify the batch size when evaluating the performance of the network on the entire training dataset.

```
# summarize performance of the model
scores = model.evaluate(X, y, batch_size=batch_size, verbose=0)
model.reset_states()
print("Model Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 34.23: Evaluate model using pre-defined batch size

Finally, you can demonstrate that the network has indeed learned the entire alphabet. You can seed it with the first letter "`A`," request a prediction, feed the prediction back in as an input, and repeat the process all the way to "`Z`."

```
# demonstrate some model predictions
seed = [char_to_int[alphabet[0]]]
for i in range(0, len(alphabet)-1):
    x = np.reshape(seed, (1, len(seed), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    print(int_to_char[seed[0]], "->", int_to_char[index])
    seed = [index]
model.reset_states()
```

Listing 34.24: Seed network and make prediction from A to Z

You can also see if the network can make predictions starting from an arbitrary letter.

```
# demonstrate a random starting point
letter = "K"
seed = [char_to_int[letter]]
print("New start: ", letter)
for i in range(0, 5):
    x = np.reshape(seed, (1, len(seed), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    print(int_to_char[seed[0]], "->", int_to_char[index])
    seed = [index]
model.reset_states()
```

Listing 34.25: Seed network with a random letter and a sequence of predictions

The entire code listing is provided below for completeness.

```python
# Stateful LSTM to learn one-char to one-char mapping
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import to_categorical
# fix random seed for reproducibility
tf.random.set_seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
    seq_in = alphabet[i:i + seq_length]
    seq_out = alphabet[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
    print(seq_in, '->', seq_out)
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (len(dataX), seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one-hot encode the output variable
y = to_categorical(dataY)
# create and fit the model
batch_size = 1
model = Sequential()
model.add(LSTM(50, batch_input_shape=(batch_size, X.shape[1], X.shape[2]), stateful=True))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
for i in range(300):
    model.fit(X, y, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
    model.reset_states()
# summarize performance of the model
scores = model.evaluate(X, y, batch_size=batch_size, verbose=0)
model.reset_states()
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
seed = [char_to_int[alphabet[0]]]
for i in range(0, len(alphabet)-1):
    x = np.reshape(seed, (1, len(seed), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    print(int_to_char[seed[0]], "->", int_to_char[index])
    seed = [index]
model.reset_states()
# demonstrate a random starting point
letter = "K"
seed = [char_to_int[letter]]
print("New start: ", letter)
for i in range(0, 5):
    x = np.reshape(seed, (1, len(seed), 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    print(int_to_char[seed[0]], "->", int_to_char[index])
    seed = [index]
model.reset_states()
```

Listing 34.26: Stateful LSTM network for one-char to one-char mapping

Running the example provides the following output.

```
Model Accuracy: 100.00%
A -> B
B -> C
C -> D
D -> E
E -> F
F -> G
G -> H
H -> I
I -> J
J -> K
K -> L
L -> M
M -> N
N -> O
O -> P
P -> Q
Q -> R
R -> S
S -> T
T -> U
U -> V
V -> W
W -> X
X -> Y
Y -> Z
New start:  K
K -> B
B -> C
C -> D
D -> E
E -> F
```

Output 34.8: Output from the stateful LSTM for one-char to one-char mapping

You can see that the network has memorized the entire alphabet perfectly. It used the context of the samples themselves and learned whatever dependency it needed to predict the next character in the sequence. You can also see that if you seed the network with the first letter, it can correctly rattle off the rest of the alphabet. You can also see that it has only learned the full alphabet sequence and from a cold start. When asked to predict the next letter from "K," it predicts "B" and falls back into regurgitating the entire alphabet. To truly predict "K," the state of the network would need to be warmed up and iteratively fed the letters from "A" to "J." This reveals that you could achieve the same effect with a *stateless* LSTM by preparing training data like this:

```
---a -> b
--ab -> c
-abc -> d
abcd -> e
```

Output 34.9: Sample of equivalent training data for non-stateful LSTM layers

Here, the input sequence is fixed at 25 (a to y to predict z) and patterns are prefixed with zero-padding. Finally, this raises the question of training an LSTM network using variable length input sequences to predict the next character.

34.7 LSTM with Variable-Length Input to One-Char Output

In the previous section, you discovered that the Keras *stateful* LSTM was really only a shortcut to replaying the first n-sequences but didn't really help us learn a generic model of the alphabet. In this section, you will explore a variation of the *stateless* LSTM that learns random subsequences of the alphabet and an effort to build a model that can be given arbitrary letters or subsequences of letters and predict the next letter in the alphabet.

Firstly, you are changing the framing of the problem. To simplify, you will define a maximum input sequence length and set it to a small value like 5 to speed up training. This defines the maximum length of subsequences of the alphabet that will be drawn for training. In extensions, this could just be set to the full alphabet (26) or longer if you allow looping back to the start of the sequence. You also need to define the number of random sequences to create — in this case, 1000. This, too, could be more or less. It's likely fewer patterns are actually required.

```python
...
# prepare the dataset of input to output pairs encoded as integers
num_inputs = 1000
max_len = 5
dataX = []
dataY = []
for i in range(num_inputs):
    start = np.random.randint(len(alphabet)-2)
    end = np.random.randint(start, min(start+max_len,len(alphabet)-1))
    sequence_in = alphabet[start:end+1]
    sequence_out = alphabet[end + 1]
    dataX.append([char_to_int[char] for char in sequence_in])
    dataY.append(char_to_int[sequence_out])
    print(sequence_in, '->', sequence_out)
```

Listing 34.27: Create dataset of variable length input sequences

Running this code in the broader context will create input patterns that look like the following:

```
PQRST -> U
W -> X
O -> P
OPQ -> R
IJKLM -> N
QRSTU -> V
ABCD -> E
X -> Y
GHIJ -> K
```

Output 34.10: Sample of variable length input sequences

The input sequences vary in length between 1 and `max_len` and therefore require zero padding. Here, use left-hand-side (prefix) padding with the Keras built-in `pad_sequences()` function.

```
...
X = pad_sequences(dataX, maxlen=max_len, dtype='float32')
```

Listing 34.28: Left-pad variable length input sequences

The trained model is evaluated on randomly selected input patterns. This could just as easily be new randomly generated sequences of characters. This could also be a linear sequence seeded with “`A`” with outputs fed back in as single character inputs. The full code listing is provided below for completeness.

```
# LSTM with Variable Length Input Sequences to One Character Output
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.sequence import pad_sequences
# fix random seed for reproducibility
np.random.seed(7)
tf.random.set_seed(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
num_inputs = 1000
max_len = 5
dataX = []
dataY = []
```

```python
for i in range(num_inputs):
    start = np.random.randint(len(alphabet)-2)
    end = np.random.randint(start, min(start+max_len,len(alphabet)-1))
    sequence_in = alphabet[start:end+1]
    sequence_out = alphabet[end + 1]
    dataX.append([char_to_int[char] for char in sequence_in])
    dataY.append(char_to_int[sequence_out])
    print(sequence_in, '->', sequence_out)
# convert list of lists to array and pad sequences if needed
X = pad_sequences(dataX, maxlen=max_len, dtype='float32')
# reshape X to be [samples, time steps, features]
X = np.reshape(X, (X.shape[0], max_len, 1))
# normalize
X = X / float(len(alphabet))
# one-hot encode the output variable
y = to_categorical(dataY)
# create and fit the model
batch_size = 1
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], 1)))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=500, batch_size=batch_size, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for i in range(20):
    pattern_index = np.random.randint(len(dataX))
    pattern = dataX[pattern_index]
    x = pad_sequences([pattern], maxlen=max_len, dtype='float32')
    x = np.reshape(x, (1, max_len, 1))
    x = x / float(len(alphabet))
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    print(seq_in, "->", result)
```

Listing 34.29: LSTM network for variable length sequence to one-char mapping

Running this code produces the following output:

```
Model Accuracy: 98.90%
['Q', 'R'] -> S
['W', 'X'] -> Y
['W', 'X'] -> Y
['C', 'D'] -> E
['E'] -> F
['S', 'T', 'U'] -> V
['G', 'H', 'I', 'J', 'K'] -> L
['O', 'P', 'Q', 'R', 'S'] -> T
['C', 'D'] -> E
['O'] -> P
```

```
['N', 'O', 'P'] -> Q
['D', 'E', 'F', 'G', 'H'] -> I
['X'] -> Y
['K'] -> L
['M'] -> N
['R'] -> T
['K'] -> L
['E', 'F', 'G'] -> H
['Q'] -> R
['Q', 'R', 'S'] -> T
```

Output 34.11: Output for the LSTM network for variable length sequences to one-char mapping

You can see that although the model did not learn the alphabet perfectly from the randomly generated subsequences, it did very well. The model was not tuned and might require more training, a larger network, or both (an exercise for the reader). This is a good natural extension to the "*all sequential input examples in each batch*" alphabet model learned above in that it can handle ad hoc queries, but this time of arbitrary sequence length (up to the max length).

34.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

APIs

LSTM layer. Keras API reference.
https://keras.io/api/layers/recurrent_layers/lstm/

34.9 Summary

In this chapter, you discovered LSTM recurrent neural networks in Keras and how they manage state. Specifically, you learned:

- ▷ How to develop a naive LSTM network for one-character to one-character prediction
- ▷ How to configure a naive LSTM to learn a sequence across time steps within a sample
- ▷ How to configure an LSTM to learn a sequence across samples by manually managing state

In the next chapter, you will put everything together to build a text generation program using LSTM.

35 Project: Text Generation with Alice in the Wonderland

Recurrent neural networks can also be used as generative models. This means that in addition to being used for predictive models (making predictions), they can learn the sequences of a problem and then generate entirely new plausible sequences for the problem domain. Generative models like this are useful not only to study how well a model has learned a problem but also to learn more about the problem domain itself.

In this chapter, you will discover how to create a generative model for text, character-by-character using LSTM recurrent neural networks in Python with Keras. After reading this chapter, you will know:

- ▷ Where to download a free corpus of text that you can use to train text generative models
- ▷ How to frame the problem of text sequences to a recurrent neural network generative model
- ▷ How to develop an LSTM to generate plausible text sequences for a given problem

Let's get started.

Note: You may want to speed up the computation for this chapter by using GPU rather than CPU hardware, such as the process described in Appendix C. This is a suggestion, not a requirement. The code will work just fine on the CPU.

Overview

This chapter is divided into five sections; they are:

- ▷ Problem Description: Project Gutenberg
- ▷ Develop a Small LSTM Recurrent Neural Network
- ▷ Generating Text with an LSTM Network
- ▷ Larger LSTM Recurrent Neural Network
- ▷ Extension Ideas to Improve the Model

35.1 Problem Description: Project Gutenberg

Many of the classical texts are no longer protected under copyright. This means you can download all the text for these books for free and use them in experiments, like creating generative models. Perhaps the best place to get access to free books that are no longer protected by copyright is Project Gutenberg. In this chapter, you will use a favorite book from childhood as the dataset: Alice's Adventures in Wonderland by Lewis Carroll[1].

You will learn the dependencies between characters and the conditional probabilities of characters in sequences so that you can, in turn, generate wholly new and original sequences of characters. This chapter is a lot of fun, and repeating these experiments with other books from Project Gutenberg is recommended. These experiments are not limited to text; you can also experiment with other ASCII data, such as computer source code, marked-up documents in LaTeX, HTML or Markdown, and more.

You can download the complete text in ASCII format (Plaintext UTF-8) for this book for free and place it in your working directory with the filename `wonderland.txt`. Now, you need to prepare the dataset ready for modeling. Project Gutenberg adds a standard header and footer to each book, which is not part of the original text. Open the file in a text editor and delete the header and footer. The header is obvious and ends with the text:

```
*** START OF THIS PROJECT GUTENBERG EBOOK ALICE'S ADVENTURES IN WONDERLAND ***
```

Output 35.1: Signal of the end of the file handler

The footer is all the text after the line of text that says:

```
THE END
```

Output 35.2: Signal of the start of the file footer

You should be left with a text file that has about 3,330 lines of text.

35.2 Develop a Small LSTM Recurrent Neural Network

In this section, you will develop a simple LSTM network to learn sequences of characters from *Alice in Wonderland*. In the next section, you will use this model to generate new sequences of characters. Let's start by importing the classes and functions you will use to train your model.

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import LSTM
from tensorflow.keras.callbacks import ModelCheckpoint
```

[1]https://www.gutenberg.org/ebooks/11

```
from tensorflow.keras.utils import to_categorical
...
```

Listing 35.1: Import classes and functions

Next, you need to load the ASCII text for the book into memory and convert all of the characters to lowercase to reduce the vocabulary the network must learn.

```
...
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
```

Listing 35.2: Load the dataset and convert to lowercase

Now that the book is loaded, you must prepare the data for modeling by the neural network. You cannot model the characters directly; instead, you must convert the characters to integers. You can do this easily by first creating a set of all of the distinct characters in the book, then creating a map of each character to a unique integer.

```
...
# create mapping of unique chars to integers
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
```

Listing 35.3: Create char-to-integer mapping

For example, the list of unique sorted lowercase characters in the book is as follows:

```
['\n', '\r', ' ', '!', '"', "'", '(', ')', '*', ',', '-', '.', ':', ';', '?', '[', ']',
 '_', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '\xbb', '\xbf', '\xef']
```

Output 35.3: List of unique characters in the dataset

You can see that there may be some characters that we could remove to further clean up the dataset to reduce the vocabulary, which may improve the modeling process. Now that the book has been loaded and the mapping prepared, you can summarize the dataset.

```
...
n_chars = len(raw_text)
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
```

Listing 35.4: Summarize the loaded dataset

Running the code to this point produces the following output.

```
Total Characters:  147674
Total Vocab:  47
```

Output 35.4: Output from summarize the dataset

You can see the book has just under 150,000 characters, and when converted to lowercase, there are only 47 distinct characters in the vocabulary for the network to learn — much more than the 26 in the alphabet. You now need to define the training data for the network. There is a lot of flexibility in how you choose to break up the text and expose it to the network during training. In this chapter, you will split the book text up into subsequences with a fixed length of 100 characters, an arbitrary length. You could just as easily split the data by sentences, padding the shorter sequences and truncating the longer ones.

Each training pattern of the network comprises 100 time steps of one character (X) followed by one character output (y). When creating these sequences, we slide this window along the whole book one character at a time, allowing each character a chance to be learned from the 100 characters that preceded it (except the first 100 characters, of course). For example, if the sequence length is 5 (for simplicity), then the first two training patterns would be as follows:

```
CHAPT -> E
HAPTE -> R
```

Output 35.5: Example of sequence construction

As you split the book into these sequences, you convert the characters to integers using the lookup table you prepared earlier.

```
...
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
    print("Total Patterns: ", n_patterns)
```

Listing 35.5: Create input/output patterns from raw dataset

Running the code to this point shows that when you split up the dataset into training data for the network to learn that you have just under 150,000 training patterns. This makes sense as, excluding the first 100 characters, you have one training pattern to predict each of the remaining characters.

```
Total Patterns:  147574
```

Output 35.6: Output summary of created patterns

Now that you have prepared your training data, you need to transform it to be suitable for use with Keras. First, you must transform the list of input sequences into the form [*samples, time steps, features*] expected by an LSTM network.

Next, you need to rescale the integers to the range 0-to-1 to make the patterns easier to learn by the LSTM network using the sigmoid activation function by default.

Finally, you need to convert the output patterns (single characters converted to integers) into a one-hot encoding. This is so that you can configure the network to predict the probability of each of the 47 different characters in the vocabulary (an easier representation) rather than trying to force it to predict precisely the next character. Each y value is converted into a sparse vector with a length of 47, full of zeros, except with a 1 in the column for the letter (integer) that the pattern represents. For example, when "n" (integer value 31) is one-hot encoded it looks as follows:

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.]
```

Output 35.7: Sample of one-hot encoded output integer

You can implement these steps as below.

```
...
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one-hot encode the output variable
y = to_categorical(dataY)
```

Listing 35.6: Prepare data ready for modeling

You can now define your LSTM model. Here, you define a single hidden `LSTM` layer with 256 memory units. The network uses dropout with a probability of 20. The output layer is a `Dense` layer using the `softmax` activation function to output a probability prediction for each of the 47 characters between 0 and 1. The problem is really a single character classification problem with 47 classes and, as such, is defined as optimizing the log loss (cross-entropy) using the Adam optimization algorithm for speed.

```
...
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 35.7: Create LSTM network to model the dataset

There is no test dataset. You are modeling the entire training dataset to learn the probability of each character in a sequence. You are not interested in the most accurate (classification accuracy) model of the training dataset. This would not be a model that predicts each character in the training dataset perfectly. Instead, you are interested in a generalization of the dataset that minimizes the chosen loss function. You are seeking a balance between generalization and overfitting but short of memorization.

The network is slow to train (about 300 seconds per epoch on an Nvidia K520 GPU). Because of the slowness and because of the optimization requirements, use model checkpointing to record all the network weights to file each time an improvement in loss is observed at the end of the epoch. You will use the best set of weights (lowest loss) to instantiate your generative model in the next section.

```
...
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss',
                             verbose=1, save_best_only=True, mode='min')
callbacks_list = [checkpoint]
```

Listing 35.8: Create checkpoints for best seen model

You can now fit your model to the data. Here, you use a modest number of 20 epochs and a large batch size of 128 patterns.

```
model.fit(X, y, epochs=20, batch_size=128, callbacks=callbacks_list)
```

Listing 35.9: Fit the model

The full code listing is provided below for completeness.

```
# Small LSTM Network to Generate Text for Alice in Wonderland
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import LSTM
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.utils import to_categorical
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
```

```
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one-hot encode the output variable
y = to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss',
                             verbose=1, save_best_only=True, mode='min')
callbacks_list = [checkpoint]
# fit the model
model.fit(X, y, epochs=20, batch_size=128, callbacks=callbacks_list)
```

Listing 35.10: Complete code listing for LSTM to model dataset

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

After running the example, you should have a number of weight checkpoint files in the local directory. You can delete them all except the one with the smallest loss value. For example, when this example was run, you can see below the checkpoint with the smallest loss that was achieved.

```
weights-improvement-19-1.9435.hdf5
```

Output 35.8: Sample of checkpoint weights for well performing model

The network loss decreased almost every epoch, so the network could likely benefit from training for many more epochs. In the next section, you will look at using this model to generate new text sequences.

35.3 Generating Text with an LSTM Network

Generating text using the trained LSTM network is relatively straightforward. Firstly, you will load the data and define the network in exactly the same way, except the network weights are loaded from a checkpoint file, and the network does not need to be trained.

Note: It seems that you might need to use the same machine/environment to generate text as was used to create the network weights (e.g., GPUs or CPUs), otherwise the network might just generate garbage.

```
...
# load the network weights
filename = "weights-improvement-19-1.9435.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 35.11: Load checkpoint network weights

Also, when preparing the mapping of unique characters to integers, you must also create a reverse mapping that you can use to convert the integers back to characters so that you can understand the predictions.

```
...
int_to_char = dict((i, c) for i, c in enumerate(chars))
```

Listing 35.12: Mapping from integers to characters

Finally, you need to actually make predictions. The simplest way to use the Keras LSTM model to make predictions is to first start with a seed sequence as input, generate the next character, then update the seed sequence to add the generated character on the end and trim off the first character. This process is repeated for as long as you want to predict new characters (e.g., a sequence of 1,000 characters in length). You can pick a random input pattern as your seed sequence, then print generated characters as you generate them.

```
...
# pick a random seed
start = np.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print("Seed:")
print("\"", ''.join([int_to_char[value] for value in pattern]), "\"")
# generate characters
for i in range(1000):
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    sys.stdout.write(result)
    pattern.append(index)
    pattern = pattern[1:len(pattern)]
print("\nDone.")
```

Listing 35.13: Seed network and generate text

The full code example for generating text using the loaded LSTM model is listed below for completeness.

```
# Load LSTM network and generate text
import sys
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import LSTM
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.utils import to_categorical
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one-hot encode the output variable
y = to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
# load the network weights (modify to your filename)
filename = "weights-improvement-19-1.9435.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
# pick a random seed
start = np.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print("Seed:")
print("\"", ''.join([int_to_char[value] for value in pattern]), "\"")
# generate characters
for i in range(1000):
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    sys.stdout.write(result)
    pattern.append(index)
    pattern = pattern[1:len(pattern)]
print("\nDone.")
```

Listing 35.14: Complete code listing for generating text for the small LSTM network

Running this example first outputs the selected random seed, then each character as it is generated. For example, below are the results from one run of this text generator. The random seed was:

```
be no mistake about it: it was neither more nor less than a pig, and she
felt that it would be quit
```

Output 35.9: Randomly selected sequence used to seed the network

The generated text with the random seed (cleaned up for presentation) was as follows.

```
be no mistake about it: it was neither more nor less than a pig, and she
felt that it would be quit e aelin that she was a little want oe toiet
ano a grtpersent to the tas a little war th tee the tase oa teettee
the had been tinhgtt a little toiee at the cadl in a long tuiee aedun
thet sheer was a little tare gereen to be a gentle of the tabdit  soenee
the gad  ouw ie the tay a tirt of toiet at the was a little
anonersen, and thiu had been woite io a lott of tueh a tiie  and taede
bot her aeain  she cere thth the bene tith the tere bane to tee
toaete to tee the harter was a little tire the same oare cade an anl ano
the garee and the was so seat the was a little gareen and the sabdit,
and the white rabbit wese tilel an the caoe and the sabbit se teeteer,
and the white rabbit wese tilel an the cade in a lonk tfne the sabdi
ano aroing to tea the was sf teet whitg the was a little tane oo thete
the sabeit  she was a little tartig to the tar tf tee the tame of the
cagd, and the white rabbit was a little toiee to be anle tite thete ofs
and the tabdit was the wiite rabbit, and
```

Output 35.10: Generated text with random seed text

Let's note some observations about the generated text.

▷ It generally conforms to the line format observed in the original text of fewer than 80 characters before a new line.

▷ The characters are separated into word-like groups, and most groups are actual English words (e.g., "the," "little," and "was"), but many are not (e.g., "lott," "tiie," and "taede").

▷ Some of the words in sequence make sense (e.g., "*and the white rabbit*"), but many do not (e.g., "*wese tilel*").

The fact that this character-based model of the book produces output like this is very impressive. It gives you a sense of the learning capabilities of LSTM networks. However, the results are not perfect. In the next section, you will look at improving the quality of results by developing a much larger LSTM network.

35.4 Larger LSTM Recurrent Neural Network

You got results, but not excellent results in the previous section. Now, you can try to improve the quality of the generated text by creating a much larger network. You will keep the number of memory units the same at 256 but add a second layer.

```
...
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 35.15: Define a stacked LSTM network

You will also change the filename of the checkpointed weights so that you can tell the difference between weights for this network and the previous (by appending the word "`bigger`" in the filename).

```
filepath="weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
```

Listing 35.16: Filename for checkpointing network weights for larger model

Finally, you will increase the number of training epochs from 20 to 50 and decrease the batch size from 128 to 64 to give the network more of an opportunity to be updated and learn. The full code listing is presented below for completeness.

```
# Larger LSTM Network to Generate Text for Alice in Wonderland
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import LSTM
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.utils import to_categorical
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
```

```
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one-hot encode the output variable
y = to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
# define the checkpoint
filepath = "weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss',
                             verbose=1, save_best_only=True, mode='min')
callbacks_list = [checkpoint]
# fit the model
model.fit(X, y, epochs=50, batch_size=64, callbacks=callbacks_list)
```

Listing 35.17: Complete code listing for the larger LSTM network

Running this example takes some time, at least 700 seconds per epoch. After running this example, you may achieve a loss of about 1.2. For example, the best result I achieved from running this model was stored in a checkpoint file with the name:

```
weights-improvement-47-1.2219-bigger.hdf5
```

Output 35.11: Sample of checkpoint weights for well performing larger model

This achieved a loss of 1.2219 at epoch 47. As in the previous section, you can use this best model from the run to generate text. The only change you need to make to the text generation script from the previous section is in the specification of the network topology and from which file to seed the network weights. The full code listing is provided below for completeness.

```
# Load Larger LSTM network and generate text
import sys
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import LSTM
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.utils import to_categorical
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename, 'r', encoding='utf-8').read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print("Total Characters: ", n_chars)
print("Total Vocab: ", n_vocab)
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
# reshape X to be [samples, time steps, features]
X = np.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one-hot encode the output variable
y = to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
# load the network weights (modify to your filename)
filename = "weights-improvement-47-1.2219-bigger.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
# pick a random seed
start = np.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print("Seed:")
print("\"", ''.join([int_to_char[value] for value in pattern]), "\"")
# generate characters
for i in range(1000):
    x = np.reshape(pattern, (1, len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = np.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in pattern]
    sys.stdout.write(result)
    pattern.append(index)
    pattern = pattern[1:len(pattern)]
print("\nDone.")
```

Listing 35.18: Complete code listing for generating text with the larger LSTM network

One example of running this text generation script produces the output below. The randomly chosen seed text was:

```
d herself lying on the bank, with her
head in the lap of her sister, who was gently brushing away s
```

Output 35.12: Randomly selected sequence used to seed the network

The generated text with the seed (cleaned up for presentation) was :

```
herself lying on the bank, with her
head in the lap of her sister, who was gently brushing away
so siee, and she sabbit said to herself and the sabbit said to herself and the sood
way of the was a little that she was a little lad good to the garden,
and the sood of the mock turtle said to herself, 'it was a little that
the mock turtle said to see it said to sea it said to sea it say it
the marge hard sat hn a little that she was so sereated to herself, and
she sabbit said to herself, 'it was a little little shated of the sooe
of the coomouse it was a little lad good to the little gooder head. and
said to herself, 'it was a little little shated of the mouse of the
good of the courte, and it was a little little shated in a little that
the was a little little shated of the thmee said to see it was a little
book of the was a little that she was so sereated to hare a little the
began sitee of the was of the was a little that she was so seally and
the sabbit was a little lad good to the little gooder head of the gad
seared to see it was a little lad good to the little good
```

Output 35.13: Generated text with random seed text

You can see that there are generally fewer spelling mistakes, and the text looks more realistic but is still quite nonsensical. For example, the same phrases get repeated again and again, like "*said to herself*" and "*little.*" Quotes are opened but not closed. These are better results, but there is still a lot of room for improvement.

35.5 Extension Ideas to Improve the Model

Below are some ideas that you may want to investigate to further improve the model:

- ▷ Predict fewer than 1,000 characters as output for a given seed
- ▷ Remove all punctuation from the source text and, therefore, from the models' vocabulary
- ▷ Try one-hot encoding for the input sequences

- ▷ Train the model on padded sentences rather than random sequences of characters
- ▷ Increase the number of training epochs to 100 or many hundreds
- ▷ Add dropout to the visible input layer and consider tuning the dropout percentage
- ▷ Tune the batch size; try a batch size of 1 as a (very slow) baseline and larger sizes from there
- ▷ Add more memory units to the layers and/or more layers
- ▷ Experiment with scale factors (temperature) when interpreting the prediction probabilities
- ▷ Change the LSTM layers to be *stateful* to maintain state across batches

35.6 Further Reading

This character text model is a popular way of generating text using recurrent neural networks. Below are some more resources and tutorials on the topic if you are interested in going deeper. Perhaps the most popular is the tutorial by Karpathy, *The Unreasonable Effectiveness of Recurrent Neural Networks*.

Papers

Ilya Sutskever, James Martens, and Geoffrey Hinton. "Generating Text with Recurrent Neural Networks". In: *Proceedings of the 28th International Conference on Machine Learning*. Bellevue, WA, USA, 2011.
https://www.cs.utoronto.ca/~ilya/pubs/2011/LANG-RNN.pdf

Articles

Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. May 2015.
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

François Chollet. *Character-level text generation with LSTM*. Keras code examples, 2020.
https://keras.io/examples/generative/lstm_character_level_text_generation/

Char RNN Example. MXNet documentation.
https://mxnet.apache.org/versions/0.11.0/tutorials/r/charRnnModel.html

Lars Eidnes. *Auto-Generating Clickbait With Recurrent Neural Networks*. 2015.
https://larseidnes.com/2015/10/13/auto-generating-clickbait-with-recurrent-neural-networks/

35.7 Summary

In this chapter, you discovered how you can develop an LSTM recurrent neural network for text generation in Python with the Keras deep learning library. After completing this chapter, you know:

- ▷ Where to download the ASCII text for classical books for free that you can use for training

- ▷ How to train an LSTM network on text sequences and how to use the trained network to generate new sequences
- ▷ How to develop stacked LSTM networks and lift the performance of the model

This marks the end of this book.

VI Appendix

A Getting More Help

This book has given you a foundation for applying deep learning in your own machine learning projects, but there is still a lot more to learn. In this chapter you will discover the places that you can go to get more help with deep learning, the Keras library as well as neural networks in general.

A.1 Artificial Neural Networks

The field of neural networks has been around for decades. As such there are a wealth of papers, books and websites on the topic. Below are a few books that I recommend if you are looking for a deeper background in the field.

- ▷ Russell D. Reed and Robert J. Marks II. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks.* MIT Press, 1999.
  https://www.amazon.com/dp/0262527014
- ▷ Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1996.
  https://www.amazon.com/dp/0198538642
- ▷ Kevin Gurney. *An Introduction to Neural Networks.* CRC Press, 1997.
  https://www.amazon.com/dp/1857285034

A.2 Deep Learning

Deep learning is a new field. Unfortunately, resources on the topic are predominately academic focused rather than practical. Nevertheless, if you are looking to go deeper into the field of deep learning, below are some suggested resources.

- ▷ Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.
  https://www.amazon.com/dp/0262035618
  (Online version at http://www.deeplearningbook.org)
  (a textbook by some of the pioneers in the field)

- ▷ Yoshua Bengio. *Learning Deep Architectures for AI.* vol. 2. Foundation and Trends in Machine Learning 1. NOW Publishers, 2009, pp. 1–127.
  https://www.iro.umontreal.ca/~lisa/pointeurs/TR1312.pdf
  (a good but academic introduction to the field)
- ▷ Jürgen Schmidhuber. "Deep Learning in Neural Networks: An Overview". *Neural Networks*, 61, Jan. 2015, pp. 85–117. DOI: https://doi.org/10.1016/j.neunet.2014.09.003.
  http://arxiv.org/abs/1404.7828
  (another excellent academic introduction paper to the field)

A.3 Python Machine Learning

Python is a growing platform for applied machine learning. The strong attraction is because Python is a fully featured programming language (unlike R) and as such you can use the same code and libraries in developing your model as you use to deploy the model into operations. The premier machine learning library in Python is scikit-learn built on top of SciPy.

- ▷ Visit the scikit-learn home page to learn more about the library and it's capabilities.
  http://scikit-learn.org
- ▷ Visit the SciPy home page to learn more about the SciPy platform for scientific computing in Python.
  http://scipy.org
- ▷ Machine Learning Mastery with Python, the precursor to this book.
  https://machinelearningmastery.com/machine-learning-with-python

If you learned some Python but not too confident, you may also find our Python book useful:

- ▷ Python for Machine Learning
  https://machinelearningmastery.com/python-for-machine-learning/

A.4 Keras Library

Keras is a fantastic but fast moving library. Large updates are still being made to the API and it is good to stay abreast of changes. You also need to know where you can ask questions to get more help with the platform.

- ▷ The Keras homepage is an excellent place to start, giving you full access to the user documentation.
  http://keras.io
- ▷ The Keras blog provides updates and tutorials on the platform.
  http://blog.keras.io
- ▷ The Keras project on GitHub hosts the code for the library. Importantly, you can use the issue tracker on the project to raise concerns and make feature requests after you have used the library.
  https://github.com/keras-team/keras

- ▷ The best place to get answers to your Keras questions is on the Google group email list.
  https://groups.google.com/g/keras-users
- ▷ A good secondary place to ask questions is Stack Overflow and use the Keras tag with your question.
  http://stackoverflow.com/questions/tagged/keras

I am always here to help if you have any questions. You can email me directly via jason@MachineLearningMastery.com and put this book title in the subject of your email.

How to Setup Python on Your Workstation

B

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda. After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning software.

B.1 Overview

In this chapter, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda
4. Install Deep Learning Libraries
5. Install Visual Studio Code environment

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, you will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python. These instructions are suitable for Windows, macOS, and Linux platforms. Windows are used below for demonstration, so you may see some Windows dialogs and file extensions.

1. Visit the Anaconda homepage `https://www.anaconda.com/`

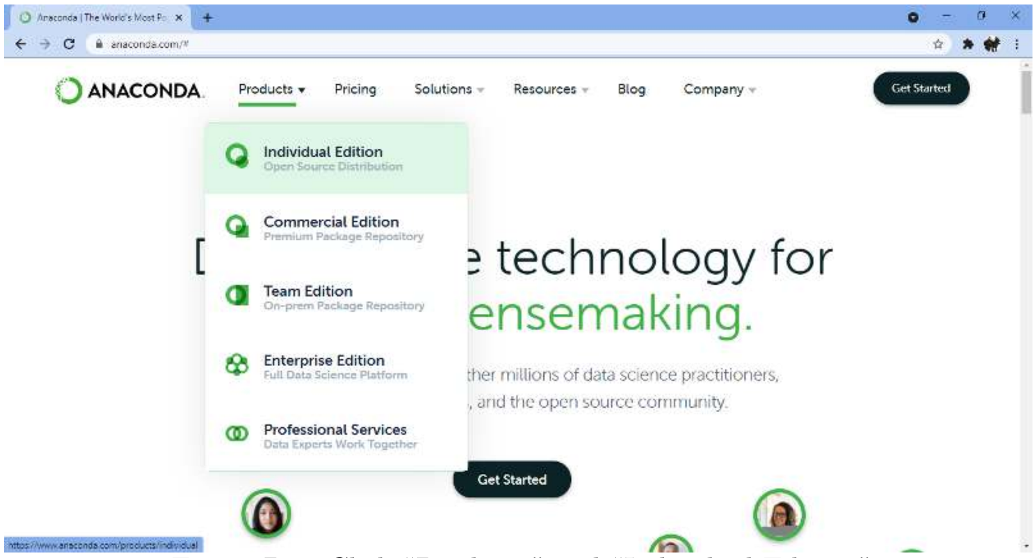
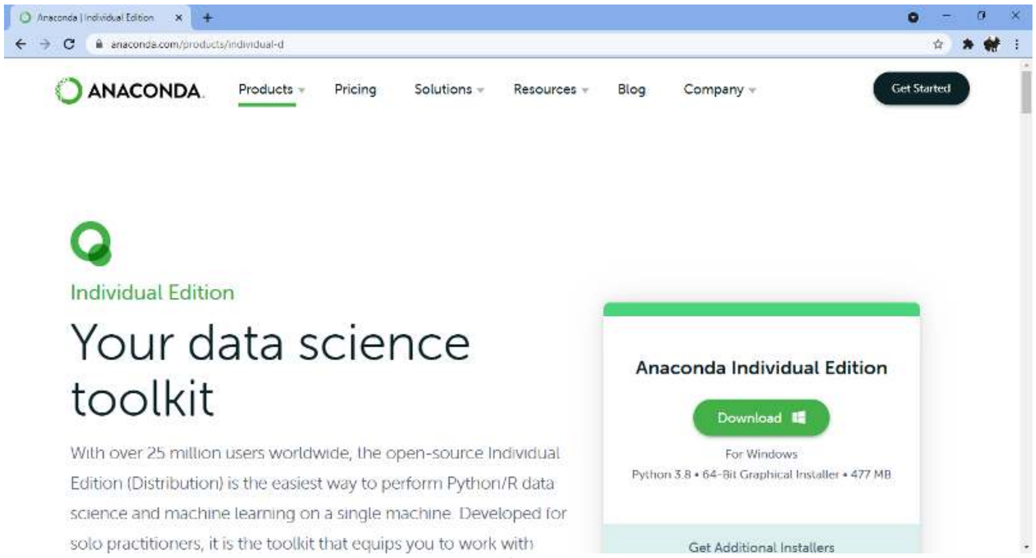


Figure B.1: Click "Products" and "Individual Edition"

2. Click "Products" from the menu and click "Individual Edition" to go to the download page `https://www.anaconda.com/products/individual-d/`.


ANACONDA
Products
Pricing
Solutions
Resources
Blog
Company
Get Started
Individual Edition
Your data science toolkit
Anaconda Individual Edition
Download
For Windows
Python 3.8 • 64-Bit Graphical Installer • 477 MB
Get Additional Installers

With over 25 million users worldwide, the open-source Individual Edition (Distribution) is the easiest way to perform Python/R data science and machine learning on a single machine. Developed for solo practitioners, it is the toolkit that equips you to work with

Figure B.2: Click Download

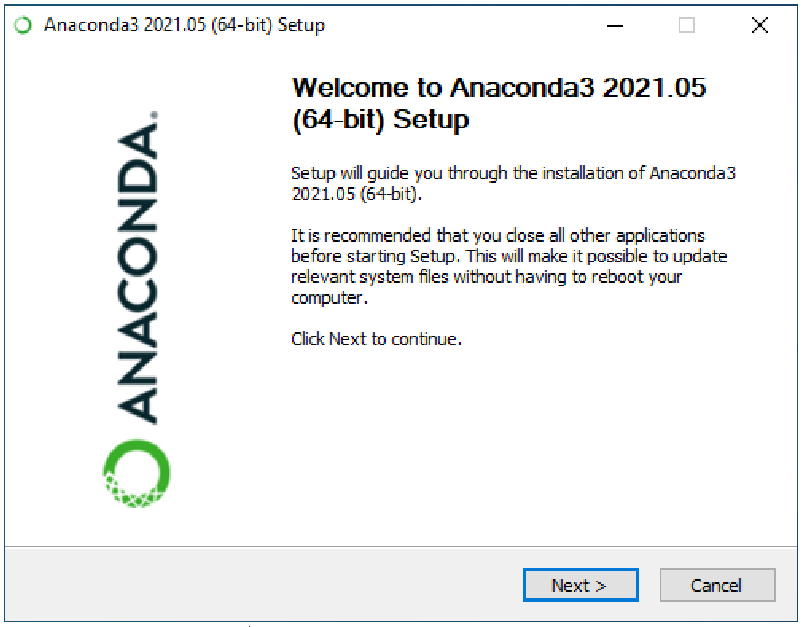
This will download the Anaconda Python package to your workstation. It will automatically give you the installer according to your OS (Windows, Linux, or MacOS). The file is about 480 MB. You should have a file with a name like:

```
Anaconda3-2021.05-Windows-x86_64.exe
```

B.3 Install Anaconda

In this step, you will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

1. Double click the downloaded file.
2. Follow the installation wizard.



Anaconda3 2021.05 (64-bit) Setup

Welcome to Anaconda3 2021.05 (64-bit) Setup

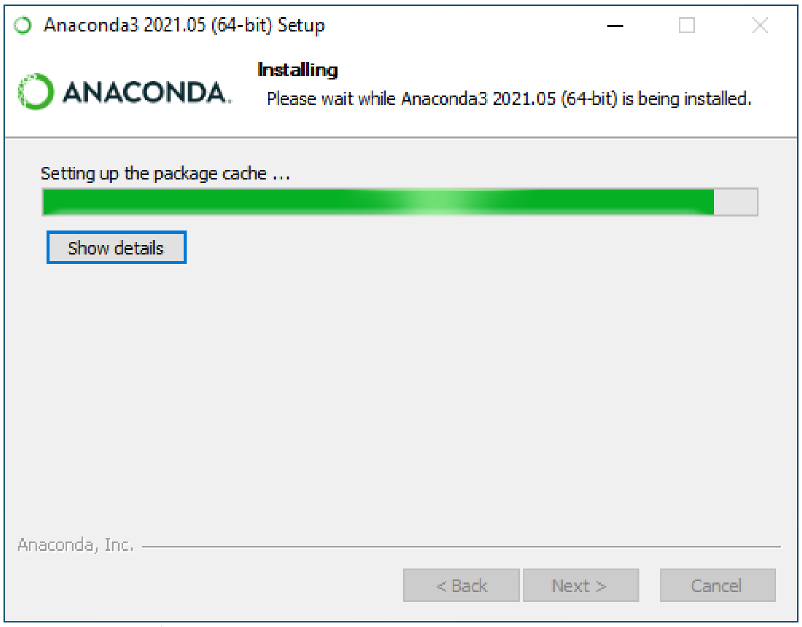
Setup will guide you through the installation of Anaconda3 2021.05 (64-bit).

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Figure B.3: Anaconda Python Installation Wizard

Installation is quick and painless. There should be no tricky questions or sticking points.



Anaconda3 2021.05 (64-bit) Setup

Installing

Please wait while Anaconda3 2021.05 (64-bit) is being installed.

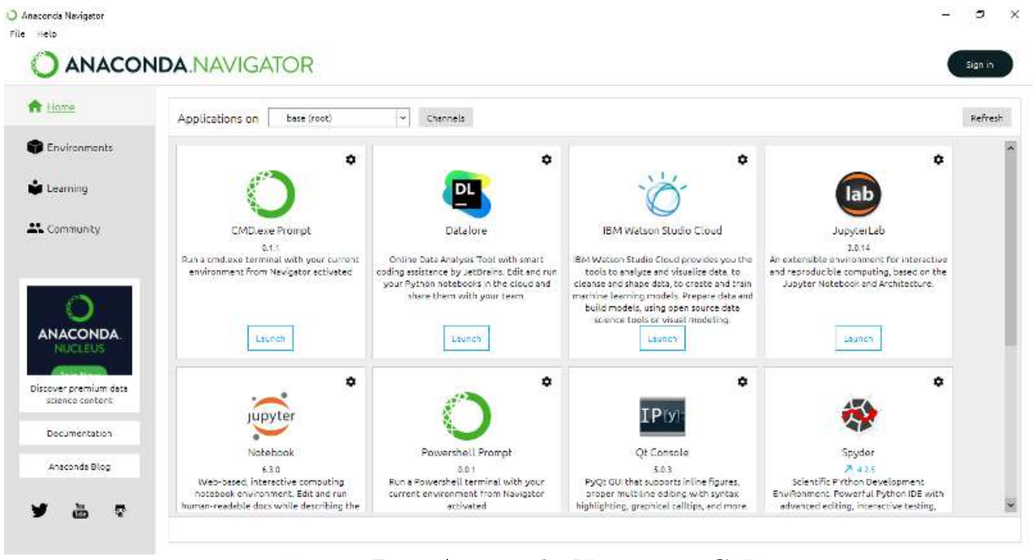
Setting up the package cache ...

Figure B.4: Anaconda Python Installation Wizard Writing Files

The installation should take less than 10 minutes and take a bit more than 5 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, you will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.



Run a cmd.exe terminal with your current environment from Navigator activated

Online Data Analysis Tool with smart coding assistance by JetBrains. Edit and run your Python notebooks in the cloud and share them with your team.

IBM Watson Studio Cloud provides you the tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Prepare data and build models, using open source data science tools or visual modeling.

An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the

Run a Powershell terminal with your current environment from Navigator activated

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing,

Figure B.5: Anaconda Navigator GUI

You can use the Anaconda Navigator and graphical development environments later; for now, it is recommended to start with the Anaconda command line environment called conda[1]. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

1. Open a terminal or CMD.exe prompt (command line window).
2. Confirm conda is installed correctly, by typing:

   ```
   conda -V
   ```

   You should see the following (or something similar):

   ```
   conda 4.10.1
   ```

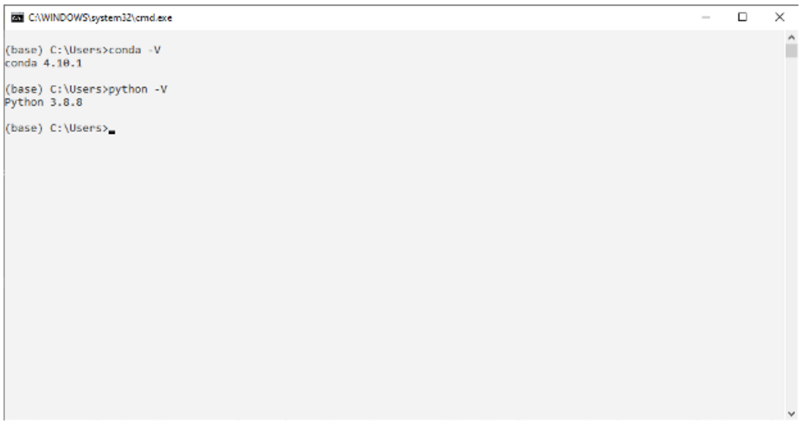
3. Confirm Python is installed correctly by typing:

   ```
   python -V
   ```

[1] https://conda.pydata.org/docs/index.html

You should see the following (or something similar):

```
Python 3.8.8
```


C:\WINDOWS\system32\cmd.exe

```
(base) C:\Users>conda -V
conda 4.10.1

(base) C:\Users>python -V
Python 3.8.8

(base) C:\Users>
```

Figure B.6: Confirm Conda and Python are Installed

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the "Further Reading" section.

4. This step is optional. You can make community supported packages available in conda by adding a "conda-forge" channel:

```
conda config --add channels conda-forge
```

5. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

You may need to install some packages and confirm the updates.

6. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type "`python`" and type the commands in directly. Alternatively, it recommended to open a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
```

```
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.1: Code to check that key Python libraries are installed.

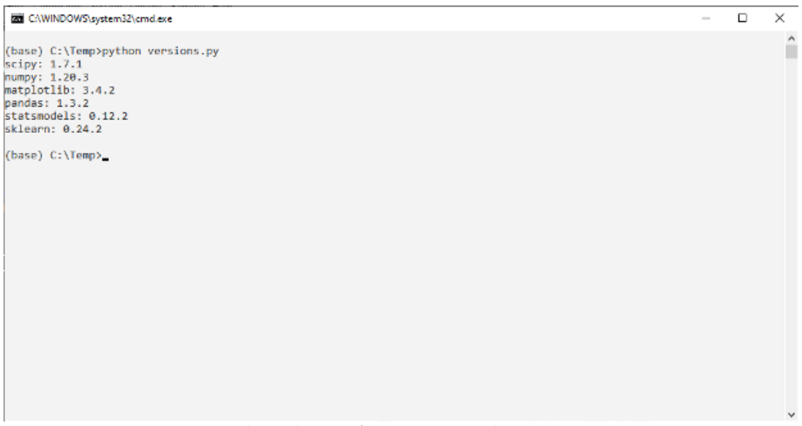
Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

You should see output like the following:

```
scipy: 1.8.1
numpy: 1.23.1
matplotlib: 3.5.1
pandas: 1.4.3
statsmodels: 0.13.2
sklearn: 1.1.1
```

Output B.1: Sample output of thhe versions script



```
(base) C:\Temp>python versions.py
scipy: 1.7.1
numpy: 1.20.3
matplotlib: 3.4.2
pandas: 1.3.2
statsmodels: 0.12.2
sklearn: 0.24.2

(base) C:\Temp>_
```

Figure B.7: Confirm Anaconda SciPy environment

B.5 Install Deep Learning Libraries

In this step, you will install Python libraries used for deep learning, specifically: TensorFlow and Keras.

1. Install the TensorFlow and Keras deep learning library by typing:

```
conda install tensorflow keras
```

2. Alternatively, you may choose to install using `pip` and a specific version of TensorFlow for your platform

```
pip install tensorflow==2.9.0
```

3. Confirm your deep learning environment is installed and working correctly.

   Create a script that prints the version numbers of each library, as you did before for the SciPy environment.

```
# check deep learning version numbers
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
# keras
import keras
print('keras: %s' % keras.__version__)
```

Listing B.2: Code to check that key deep learning libraries are installed.

Save the script to a file `deep_versions.py`. Run the script by typing:

```
python deep_versions.py
```

Output B.2: Run script from the command line.

You should see output like:

```
tensorflow: 2.9.2
keras: 2.9.0
```

Output B.3: Sample output of the deep learning versions script

B.6 Install Visual Studio Code for Python

If you have installed Anaconda, you will have an IDE called Spyder installed. You can develop your Python project with Spyder. The other popular way of writing Python code nowadays is to use Visual Studio Code. It is a free programming environment from Microsoft. Visual Studio Code can do a lot of things via "extensions". Python extension is what you should get.

These instructions are suitable for Windows, macOS, and Linux platforms. Below, macOS is used to demonstrate, so you may see some Windows dialogs and file extensions.

1. Visit the Visual Studio Code homepage `https://code.visualstudio.com`

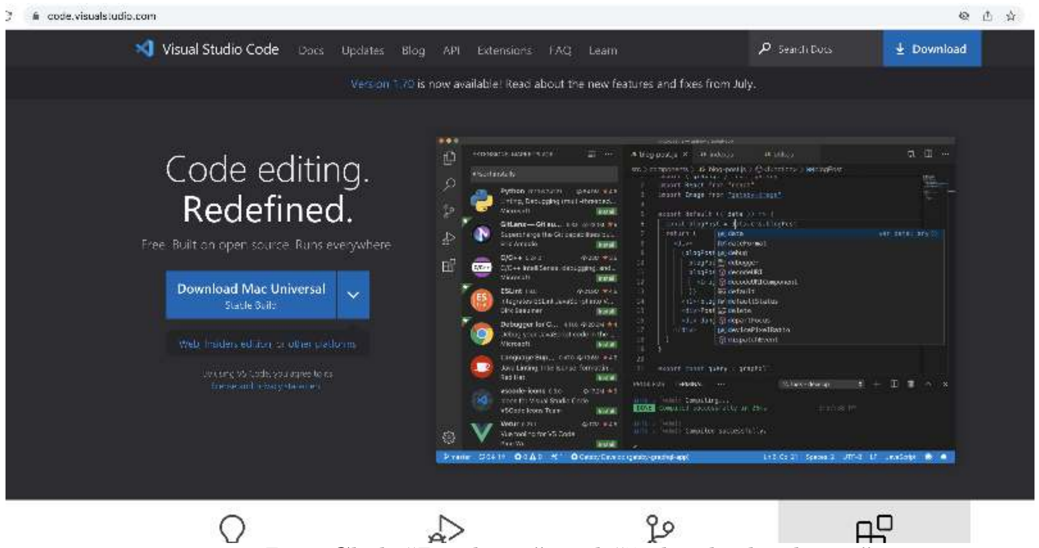

code.visualstudio.com
Visual Studio Code
Docs
Updates
Blog
API
Extensions
FAQ
Learn
Search Docs
Download
Version 1.70 is now available! Read about the new features and fixes from July.
Code editing. Redefined.
Free. Built on open source. Runs everywhere.
Download Mac Universal
Stable Build
Web, Insiders edition, or other platforms

Figure B.8: Click "Products" and "Individual Edition"

2. Click the download button at the top toolbar or at the center of the screen to download a ZIP file (such as `VSCode-darwin-universal.zip`). It is around 200MB in size
3. Expand the ZIP you will find the application program. In macOS, you should move it into your `/Applications` folder

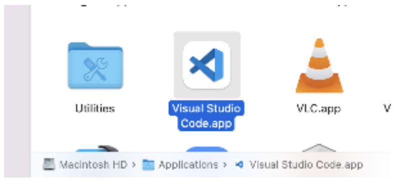
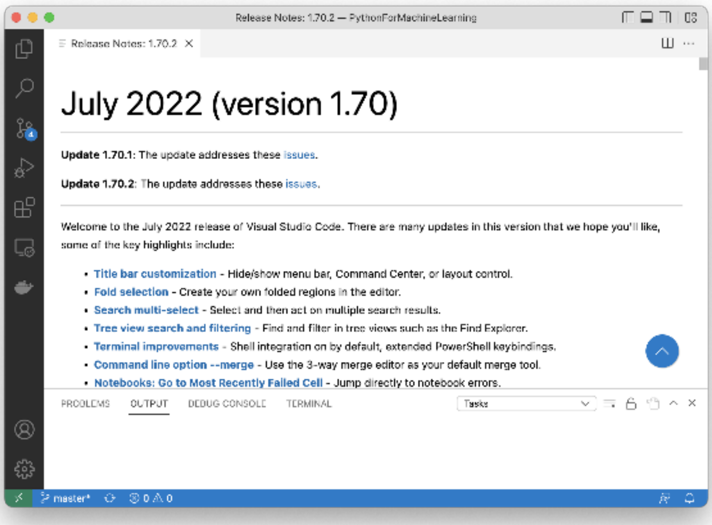


Figure B.9: Visual Studio Code in `/Applications` in macOS

4. Running Visual Studio Code will show you the main screen like the following:


Release Notes: 1.70.2 — PythonForMachineLearning

Release Notes: 1.70.2

July 2022 (version 1.70)

Update 1.70.1: The update addresses these issues.

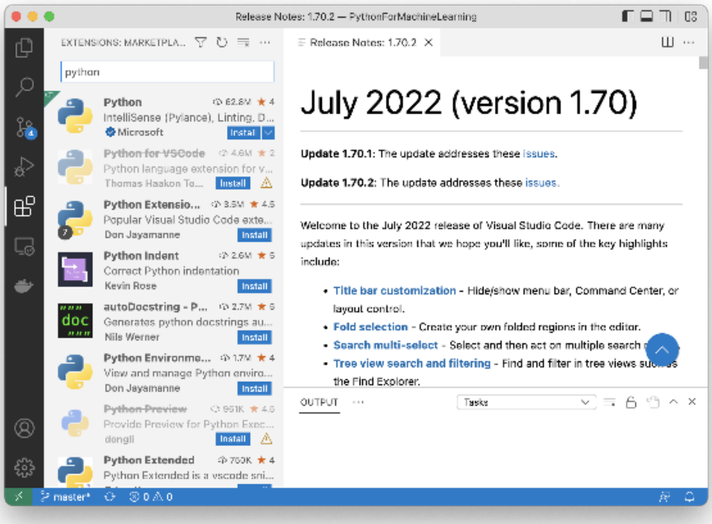
Update 1.70.2: The update addresses these issues.

Welcome to the July 2022 release of Visual Studio Code. There are many updates in this version that we hope you'll like, some of the key highlights include:

- **Title bar customization** - Hide/show menu bar, Command Center, or layout control.
- **Fold selection** - Create your own folded regions in the editor.
- **Search multi-select** - Select and then act on multiple search results.
- **Tree view search and filtering** - Find and filter in tree views such as the Find Explorer.
- **Terminal improvements** - Shell integration on by default, extended PowerShell keybindings.
- **Command line option --merge** - Use the 3-way merge editor as your default merge tool.
- **Notebooks: Go to Most Recently Failed Cell** - Jump directly to notebook errors.

Figure B.10: Visual Studio Code main screen

You can click on the building block icon at left toolbar to open the extensions marketplace. Typing "`python`" on the search box will usually show the Microsoft-developed Python extension at top.


Release Notes: 1.70.2 — PythonForMachineLearning

EXTENSIONS: MARKETPLA...

python

Python — 62.8M ★ 4 — IntelliSense (Pylance), Linting, D... — Microsoft — Install

~~Python for VSCode~~ — 4.6M ★ 2 — Python language extension for v... — Thomas Haakon To... — Install

Python Extensio... — 3.5M ★ 4.5 — Popular Visual Studio Code exte... — Don Jayamanne — Install

Python Indent — 2.6M ★ 5 — Correct Python indentation — Kevin Rose — Install

autoDocstring - P... — 2.7M ★ 5 — Generates python docstrings au... — Nils Werner — Install

Python Environme... — 1.7M ★ 4 — View and manage Python enviro... — Don Jayamanne — Install

~~Python Preview~~ — 951K ★ 4.5 — Provide Preview for Python Exec... — dongli — Install

Python Extended — 700K ★ 4 — Python Extended is a vscode sni...

Release Notes: 1.70.2

July 2022 (version 1.70)

Update 1.70.1: The update addresses these issues.

Update 1.70.2: The update addresses these issues.

Welcome to the July 2022 release of Visual Studio Code. There are many updates in this version that we hope you'll like, some of the key highlights include:

- **Title bar customization** - Hide/show menu bar, Command Center, or layout control.
- **Fold selection** - Create your own folded regions in the editor.
- **Search multi-select** - Select and then act on multiple search
- **Tree view search and filtering** - Find and filter in tree views su the Find Explorer.

Figure B.11: Searching for Python extension in the extensions marketplace

5. Click on the "Install' button on the extension marketplace will get the extension installed. It should be very quick.

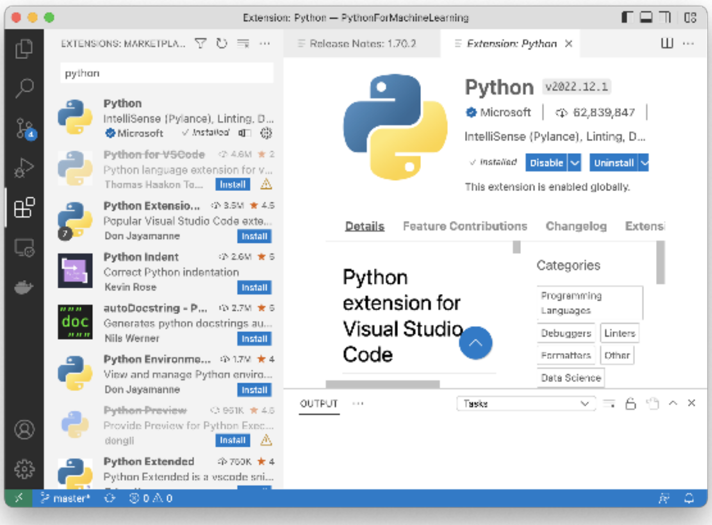


Figure B.12: Python extension installed in Visual Studio Code

Afterwards, you will find your Visual Studio Code can launch Python interpreter or Python debugger to run your program.

B.7 Further Reading

This section provides resources if you want to know more about Anaconda.

▷ Anaconda Documentation
`https://docs.continuum.io/`

▷ Anaconda Documentation: Installation
`https://docs.continuum.io/anaconda/install`

▷ Anaconda Navigator
`https://docs.continuum.io/anaconda/navigator.html`

▷ The conda command line tool
`https://conda.pydata.org/docs/index.html`

▷ Using conda
`https://conda.pydata.org/docs/using/`

▷ conda-forge
`https://conda-forge.org//docs/using/`

▷ Instructions for installing TensorFlow in Anaconda
`https://docs.anaconda.com/anaconda/user-guide/tasks/tensorflow/`

B.8 Summary

Congratulations, you now have a working Python development environment for machine learning. You can now learn and practice machine learning on your workstation.

How to Setup Amazon EC2 for Deep Learning on GPUs

C

Large deep learning models require a lot of compute time to run. You can run them on your CPU but it can take hours or days to get a result. If you have access to a GPU on your desktop, you can drastically speed up the training time of your deep learning models. In this project you will discover how you can get access to GPUs to speed up the training of your deep learning models by using the Amazon Web Service (AWS) infrastructure. For less than a dollar per hour and often a lot cheaper you can use this service from your workstation or laptop. After working through this project you will know:

▷ How to create an account and log-in to Amazon Web Service.

▷ How to launch a server instance for deep learning.

▷ How to configure a server instance for faster deep learning on the GPU.

Let's get started.

C.1 Overview

The process is quite simple because most of the work has already been done for us. Below is an overview of the process.

▷ Setup Your AWS Account.

▷ Launch Your Server Instance.

▷ Login and Run Your Code.

▷ Close Your Server Instance.

Note: It costs money to use a virtual server instance on Amazon. The cost is low for model development (e.g., less than one US dollar per hour), which is why this is so attractive, but it is not free. The server instance runs Linux. It is desirable although not required that you know how to navigate Linux or a Unix-like environment. We're just running our Python scripts, so no advanced skills are needed.

Note: The specific versions may differ as the software and libraries are updated frequently.

C.2 Setup Your AWS Account

You need an account on Amazon Web Services (https://aws.amazon.com).

1. Go to https://aws.amazon.com. You should click "Sign In" at the toolbar

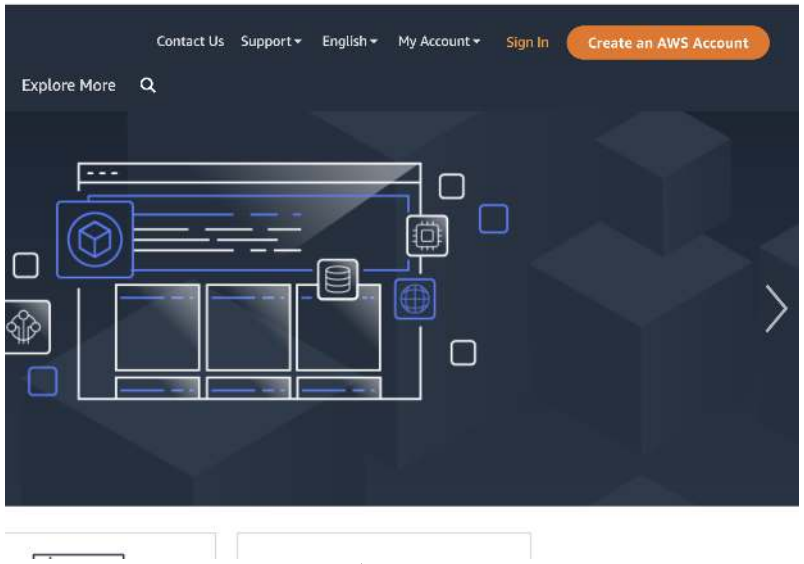
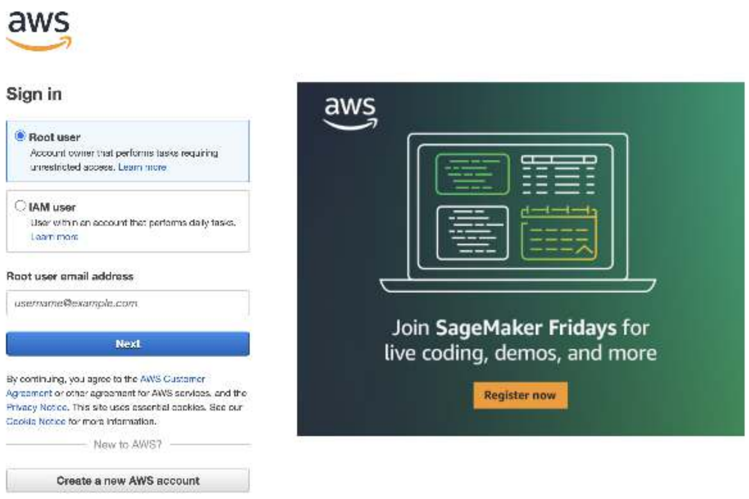


Figure C.1: AWS Sign In Button

2. This will bring you to the following screen. You can enter your login credentials, or you can sign up an account if you didn't have one yet.



Sign in

[✓ Root user]
Account owner that performs tasks requiring unrestricted access. Learn more

IAM user
User within an account that performs daily tasks. Learn more

Root user email address

username@example.com

Next

By continuing, you agree to the AWS Customer Agreement or other agreement for AWS services, and the Privacy Notice. This site uses essential cookies. See our Cookie Notice for more information.

New to AWS?

Create a new AWS account

Figure C.2: Login screen for AWS console

Once you have an account you can log into the Amazon Web Services console. You will see a range of different services that you can access.

C.3 Launch Your Server Instance

Now that you have an AWS account, you want to launch an EC2 virtual server instance on which you can run Keras. Launching an instance is as easy as selecting the image to load and starting the virtual server. Thankfully there is already an image available that has almost everything we need it is called the "*Deep Learning AMI GPU TensorFlow 2.9.1 (Ubuntu 20.04) 20220816*" and was created and is maintained by Amazon. Let's launch it as an instance.

1. Login to your AWS console if you have not already.
   `https://console.aws.amazon.com/console/home`
2. At the toolbar, you can select your server location. The server will cost differently in different region. Then click on EC2 for launching a new virtual server.

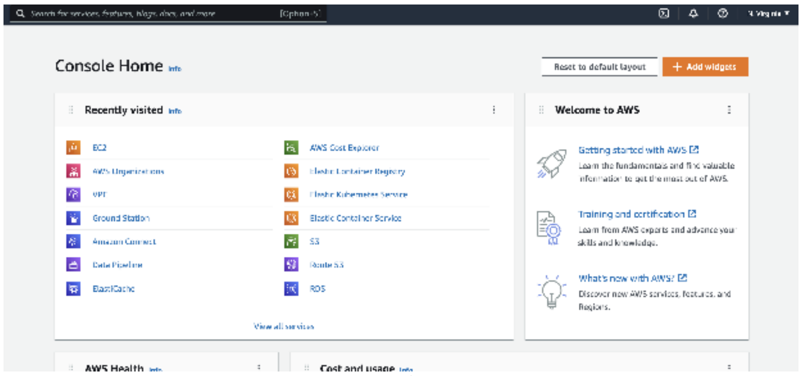
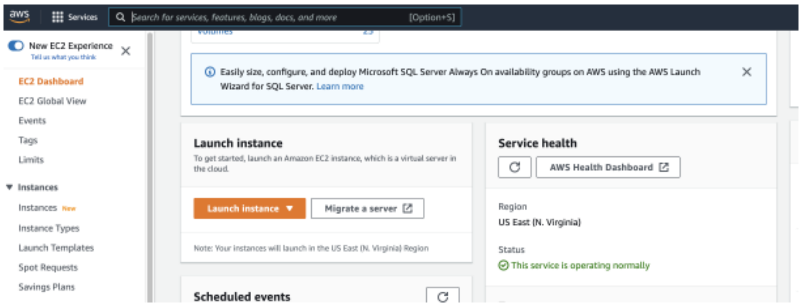

Console Home Info
Reset to default layout
+ Add widgets
Recently visited Info
EC2
AWS Organizations
VPC
Ground Station
Amazon Connect
Data Pipeline
ElastiCache
AWS Cost Explorer
Elastic Container Registry
Elastic Kubernetes Service
Elastic Container Service
S3
Route 53
RDS
View all services
Welcome to AWS
Getting started with AWS
Learn the fundamentals and find valuable information to get the most out of AWS.
Training and certification
Learn from AWS experts and advance your skills and knowledge.
What's new with AWS?
Discover new AWS services, features, and Regions.
AWS Health Info
Cost and usage Info

Figure C.3: AWS console

3. Click the *Launch instance* button.



Easily size, configure, and deploy Microsoft SQL Server Always On availability groups on AWS using the AWS Launch Wizard for SQL Server. Learn more

To get started, launch an Amazon EC2 instance, which is a virtual server in the cloud.

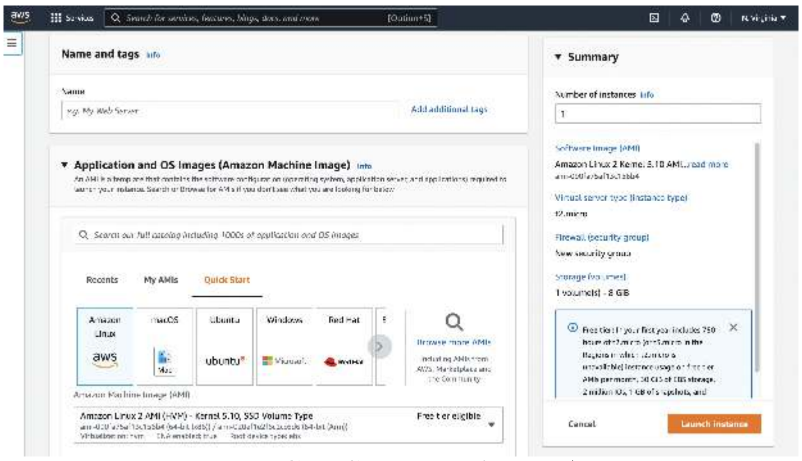
Note: Your instances will launch in the US East (N. Virginia) Region

Region
US East (N. Virginia)

Status
This service is operating normally

Figure C.4: Launch an EC2 instance

4. Give a name to your instance. Then click on *Browse more AMIs* and search with the keyword "deep learning". An AMI is an Amazon Machine Image. It is a frozen instance of a server that you can select and instantiate on a new virtual server.



Name and tags

Name: e.g. My Web Server

Application and OS Images (Amazon Machine Image)

Search our full catalog including 1000s of application and OS images

Recents | My AMIs | Quick Start

Amazon Machine Image (AMI): Amazon Linux 2 AMI (HVM) - Kernel 5.10, SSD Volume Type — Free tier eligible

Summary

Number of instances: 1

Software Image (AMI): Amazon Linux 2 Kernel 5.10 AMI...

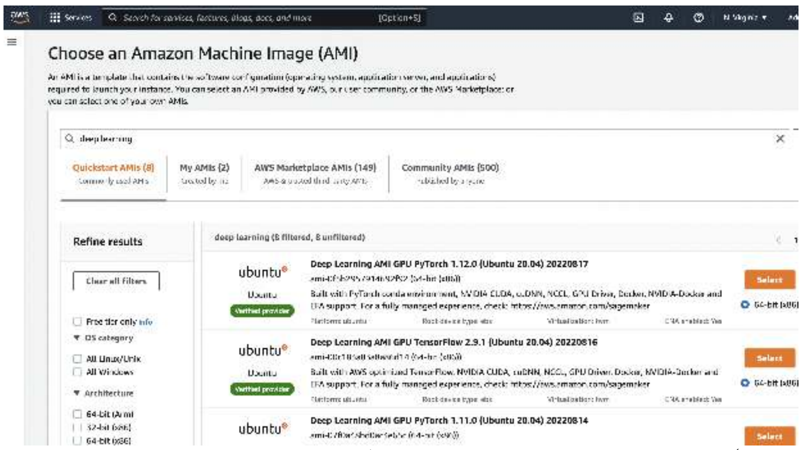
Virtual server type (instance type): t2.micro

Firewall (security group): New security group

Storage (volumes): 1 volume(s) - 8 GiB

Figure C.5: Searching for an AMI

5. We will select the one with TensorFlow pre-installed.



Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

Search: deep learning

Quickstart AMIs (8) | My AMIs (2) | AWS Marketplace AMIs (149) | Community AMIs (500)

deep learning (8 filtered, 8 unfiltered)

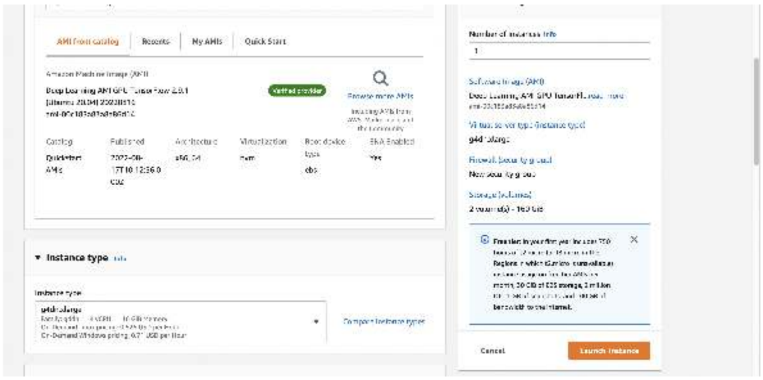
Deep Learning AMI GPU PyTorch 1.12.0 (Ubuntu 20.04) 20220817
Built with PyTorch conda environment, NVIDIA CUDA, cuDNN, NCCL, GPU Driver, Docker, NVIDIA-Docker and EFA support. For a fully managed experience, check: https://aws.amazon.com/sagemaker

Deep Learning AMI GPU TensorFlow 2.9.1 (Ubuntu 20.04) 20220816
Built with AWS optimized TensorFlow, NVIDIA CUDA, cuDNN, NCCL, GPU Driver, Docker, NVIDIA-Docker and EFA support. For a fully managed experience, check: https://aws.amazon.com/sagemaker

Deep Learning AMI GPU PyTorch 1.11.0 (Ubuntu 20.04) 20220814

Figure C.6: Selecting "Deep Learning AMI GPU TensorFlow 2.9.1 (Ubuntu 20.04)"

6. Now you need to select the hardware on which to run the image. The drop down box will show all the available hardware that are compatible to your AMI. The instance types begin with a "`g`" comes with GPU and those begin with a "`p`" includes a Tesla core GPU, which are much faster. In below, `g4dn.xlarge` is selected but you can pick one with suitable amount of memory, disk size, and cost for your project.



Instance type: g4dn.xlarge

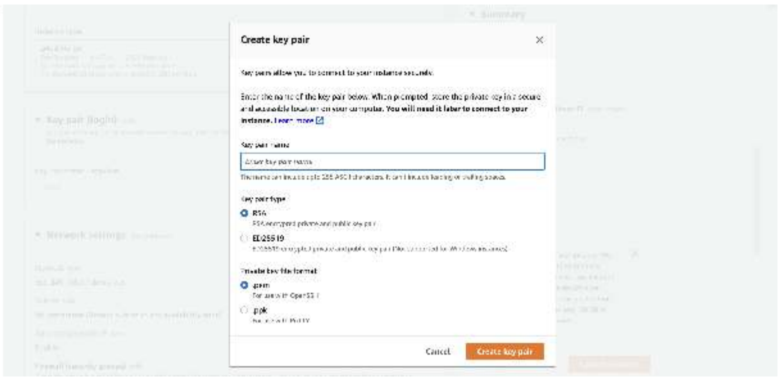
Number of instances: 1

Storage (volumes): 2 volume(s)

Figure C.7: Selecting instance type `g4ad.xlarge` for the new server

You can learn more about EC2 instance types at https://aws.amazon.com/ec2/instance-types/.

7. You need to select a key pair for your machine. It is used to login to the server via SSH using key-based authentication. You may select an existing one, or create a new key pair.



Create key pair

Key pairs allow you to connect to your instance securely.

Enter the name of the key pair below. When prompted, store the private key in a secure and accessible location on your computer. You will need it later to connect to your instance. Learn more

Key pair name

Enter key pair name

The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

Key pair type

[✓ RSA] RSA encrypted private and public key pair

ED25519 ED25519 encrypted private and public key pair (Not supported for Windows instances)

Private key file format

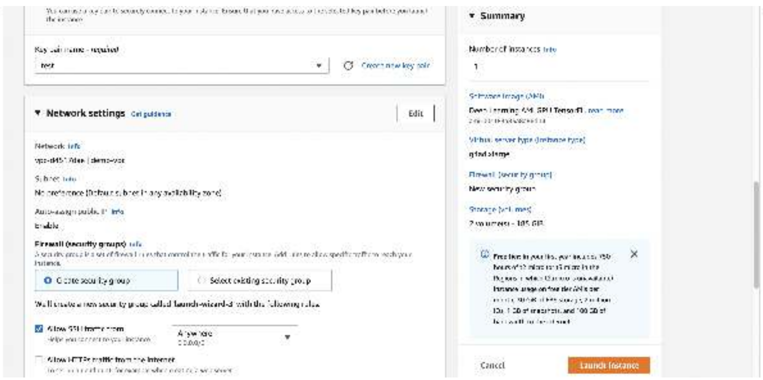
[✓ .pem] For use with OpenSSH

.ppk For use with PuTTY

Figure C.8: Creating a new key pair

Depends on the program you use as your SSH client, you may select to download your key between PEM format or PPK format. Keep this key file safe. You will not be able to get it again after this step.

8. As a final check, you should see your instance with "Auto-assign public IP" enabled and "Allow SSH traffic from" set to "Anywhere" so we can access to the instance after launch. Then you can click *Launch* to create your instance.



Key pair name - required: test

▾ Network settings

Network: vpc-d451 | demo-vpc

Subnet: No preference (Default subnet in any availability zone)

Auto-assign public IP: Enable

Firewall (security groups)

[✓ Create security group] Select existing security group

We'll create a new security group called launch-wizard-3 with the following rules:

[✓ Allow SSH traffic from] Anywhere 0.0.0.0/0

Allow HTTPs traffic from the internet

▾ Summary

Number of instances: 1

Software Image (AMI): Deep Learning AMI GPU TensorFl...

Virtual server type (instance type): g4dn.xlarge

Firewall (security group): New security group

Storage (volumes): 2 volume(s) - 105 GiB

Figure C.9: Launch a new instance

Your server is now running and ready for you to log in.

C.4 Login, Configure and Run

Now that you have launched your server instance, it is time to log in and start using it.

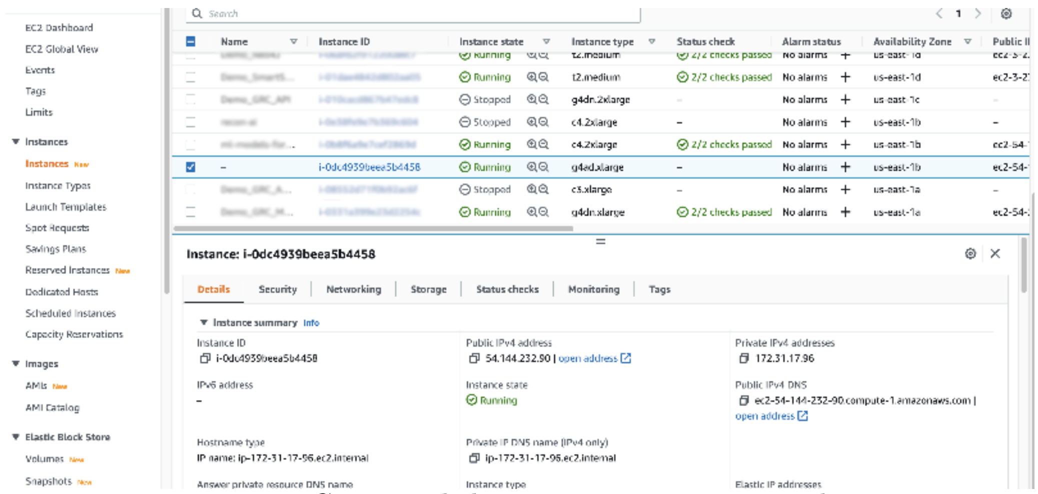
1. Open a Terminal and change directory to where you downloaded your key pair.

2. If you have not already done so, restrict the access permissions on your key pair file. This is required as part of the SSH access to your server. For example, open a terminal on your workstation and type:

```
cd Downloads
chmod 600 my-aws-key.pem
```

Listing C.1: Change permissions of your key file

3. Click *Instances* in your Amazon EC2 console if you have not done so already. And find and select the instance that you just created.



	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public I
	[redacted]	[redacted]	Running	t2.medium	2/2 checks passed	No alarms +	us-east-1d	ec2-3-2
	[redacted]	[redacted]	Running	t2.medium	2/2 checks passed	No alarms +	us-east-1d	ec2-3-2
	[redacted]	[redacted]	Stopped	g4dn.2xlarge	–	No alarms +	us-east-1c	–
	[redacted]	[redacted]	Stopped	c4.2xlarge	–	No alarms +	us-east-1b	–
	[redacted]	[redacted]	Running	c4.2xlarge	2/2 checks passed	No alarms +	us-east-1b	ec2-54-
✓	–	i-0dc4939beea5b4458	Running	g4ad.xlarge	–	No alarms +	us-east-1b	ec2-54-
	[redacted]	[redacted]	Stopped	c5.xlarge	–	No alarms +	us-east-1a	–
	[redacted]	[redacted]	Running	g4dn.xlarge	2/2 checks passed	No alarms +	us-east-1a	ec2-54-

Instance: i-0dc4939beea5b4458

Instance summary Info

Instance ID i-0dc4939beea5b4458	Public IPv4 address 54.144.232.90 | open address	Private IPv4 addresses 172.31.17.96
IPv6 address –	Instance state Running	Public IPv4 DNS ec2-54-144-232-90.compute-1.amazonaws.com | open address
Hostname type IP name: ip-172-31-17-96.ec2.internal	Private IP DNS name (IPv4 only) ip-172-31-17-96.ec2.internal	
Answer private resource DNS name	Instance type	Elastic IP addresses

Figure C.10: Find the instance you just created

4. Copy the *Public IPv4 address* (at first row of the instance detail table) to your clipboard. In this example my IP address is `54.144.232.90`. **Do not use this IP address, it will not work as your server IP address will be different**.

5. Open a Terminal and change directory to where you downloaded your key pair. Login to your server using SSH, for example:

```
ssh -i my-aws-key.pem ubuntu@54.144.232.90
```

Listing C.2: Log-in To Your AWS Instance.

If prompted, type `yes` and press enter.

In the SSH command, we use the syntax `username@address` to specify the server address and the login name. The login name depends on the AMI you used. If you used Ubuntu AMI, it is `ubuntu`. But if you used Amazon Linux AMI, it would be `ec2-user`. You can see these details at `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/connection-prereqs.html`

6. If your SSH session is established, you should see the login screen as follows:

```
=============================================================================
       __|  __|_  )
       _|  (     /   Deep Learning AMI GPU TensorFlow 2.9.1 (Ubuntu 20.04)
      ___|\___|___|
=============================================================================

Welcome to Ubuntu 20.04.4 LTS (GNU/Linux 5.15.0-1017-aws x86_64v)


TensorFlow 2.9.1 and utility libraries are installed in /usr/bin/python3.9.
To access them, use /usr/bin/python3.9.

NVIDIA driver version: 510.47.03
CUDA version: 11.2

AWS Deep Learning AMI Homepage: https://aws.amazon.com/machine-learning/amis/
Release Notes: https://docs.aws.amazon.com/dlami/latest/devguide/appendix-ami-release-notes.html
Support: https://forums.aws.amazon.com/forum.jspa?forumID=263
For a fully managed experience, check out Amazon SageMaker at https://aws.amazon.com/sagemaker
Security scan reports for python packages are located at: /opt/aws/dlami/info/
=============================================================================

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Mon Aug 22 20:44:29 UTC 2022


0 updates can be applied immediately.



The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

ubuntu@ip-172-31-17-96:~$
```

Figure C.11: Login screen for your AWS server

You are now logged into your server. You are now free to run your code.

Depends on the AMI, you may need to launch Python differently. In this particular AMI, there are Python 3.8 and 3.9 installed but all deep learning libraries are available on Python 3.9 only. You need to use `python3.9` command instead of `python` or `python3` for your projects.

C.5 Build and Run Models on AWS

This section offers some tips for running your code on AWS.

C.5.1 Copy Scripts and Data to AWS

You can get started quickly by copying your files to your running AWS instance. For example, you can copy the examples provided with this book to your AWS instance using the `scp` command as follows:

```
scp -i my-aws-key.pem -r src ubuntu@54.144.232.90:~/
```

Listing C.3: Example for copying sample code to AWS

This will copy the entire `src/` directory to your home directory on your AWS instance. You can easily adapt this example to get your larger datasets from your workstation onto your AWS instance. Note that Amazon may impose charges for moving very large amounts of data in and out of your AWS instance. Refer to Amazon documentation for relevant charges.

C.5.2 Run Models on AWS

You can run your scripts on your AWS instance as per normal:

```
python3.9 filename.py
```

Listing C.4: Example of running a Python script on AWS

In other instances, you may use another command for the Python interpreter, such as `python3`. You are using AWS to create large neural network models that may take hours or days to train. As such, it is a better idea to run your scripts as a background job. This allows you to close your terminal and your workstation while your AWS instance continues to run your script. You can easily run your script as a background process as follows:

```
nohup /path/to/script >/path/to/script.log 2>&1 < /dev/null &
```

Listing C.5: Run script as a background process.

You can then check the status and results in your `script.log` file later.

C.6 Close Your EC2 Instance

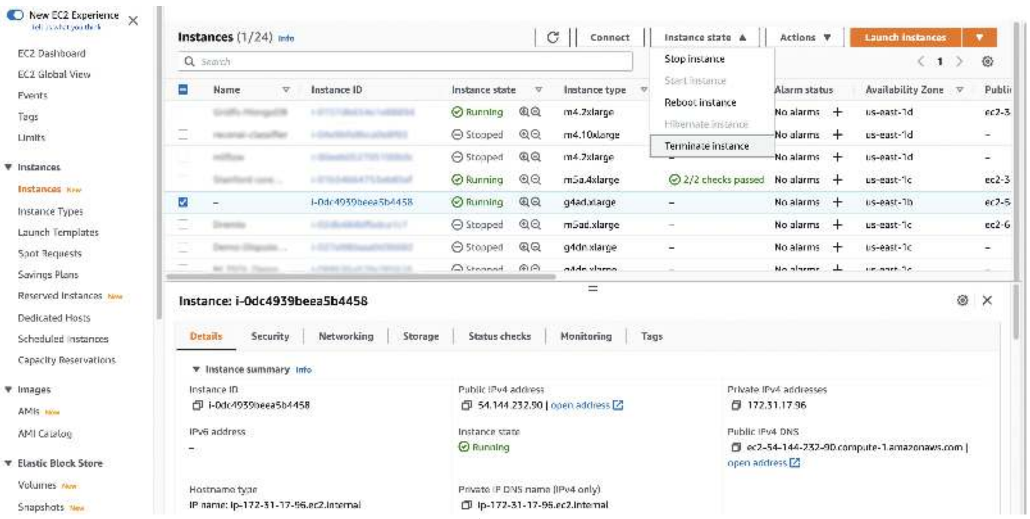
When you are finished with your work you must terminate your instance. Remember you are charged by the amount of time that you use the instance. It is cheap, but you do not want to leave an instance on if you are not using it.

1. Log out of your instance at the terminal, for example you can type:

   ```
   exit
   ```

   Listing C.6: Command to log-out of server instance

2. Log in to your AWS account with your web browser.
3. Click EC2.
4. Click *Instances* from the left-hand side menu, then find and select your instance.
5. At the toolbar, click *Instance state* and select *Terminate instance*. Confirm that you want to terminate your running instance.



Instances (1/24)

Name	Instance ID	Instance state	Instance type		Alarm status	Availability Zone	Publi
[redacted]	[redacted]	Running	m4.2xlarge		No alarms	us-east-1d	ec2-3
[redacted]	[redacted]	Stopped	m4.10xlarge		No alarms	us-east-1d	–
[redacted]	[redacted]	Stopped	m4.2xlarge	–	No alarms	us-east-1d	–
[redacted]	[redacted]	Running	m5a.4xlarge	2/2 checks passed	No alarms	us-east-1c	ec2-3
–	i-0dc4939beea5b4458	Running	g4ad.xlarge	–	No alarms	us-east-1b	ec2-5
[redacted]	[redacted]	Stopped	m5ad.xlarge	–	No alarms	us-east-1c	ec2-6
[redacted]	[redacted]	Stopped	g4dn.xlarge	–	No alarms	us-east-1c	–

Instance: i-0dc4939beea5b4458

Instance summary Info

Instance ID: i-0dc4939beea5b4458
Public IPv4 address: 54.144.232.90 | open address
Private IPv4 addresses: 172.31.17.96
IPv6 address: –
Instance state: Running
Public IPv4 DNS: ec2-54-144-232-90.compute-1.amazonaws.com | open address
Hostname type: IP name: ip-172-31-17-96.ec2.internal
Private IP DNS name (IPv4 only): ip-172-31-17-96.ec2.internal

Figure C.12: AWS Sign In Button

There is a *Stop instance* option. That will only stop your instance from running but still occupies the resources such as storage and IP address. Hence terminate is how you stop AWS from billing you for the server.

It may take a number of seconds for the instance to close and to be removed from your list of instances.

C.7 Tips and Tricks for Using Keras on AWS

Below are some tips and tricks for getting the most out of using Keras on AWS instances.

- ▷ **Design a suite of experiments to run beforehand**. Experiments can take a long time to run and you are paying for the time you use. Make time to design a batch of experiments to run on AWS. Put each in a separate file and call them in turn from another script. This will allow you to answer multiple questions from one long run, perhaps overnight.
- ▷ **Always close your instance at the end of your experiments**. You do not want to be surprised with a very large AWS bill.
- ▷ **Try spot instances for a cheaper but less reliable option**. Amazon sell unused time on their hardware at a much cheaper price, but at the cost of potentially having your instance closed at any second. If you are learning or your experiments are not critical, this might be an ideal option for you. You can access spot instances from the *Spot Instance* option on the left hand side menu in your EC2 web console.

C.8 Further Reading

Below is a list of resources to learn more about AWS and developing deep learning models in the cloud.

- ▷ An introduction to Amazon Elastic Compute Cloud (EC2).
  http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- ▷ An introduction to Amazon Machine Images (AMI).
  http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html
- ▷ AWS instance types.
  https://aws.amazon.com/ec2/instance-types/

C.9 Summary

In this lesson you discovered how you can develop and evaluate your large deep learning models in Keras using GPUs on the Amazon Web Service. You learned:

- ▷ Amazon Web Services with their Elastic Compute Cloud offers an affordable way to run large deep learning models on GPU hardware.
- ▷ How to setup and launch an EC2 server for deep learning experiments.
- ▷ How to update the Keras version on the server and confirm that the system is working correctly.
- ▷ How to run Keras experiments on AWS instances in batch as background tasks.

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

- ▷ You started off with an interest in deep learning and a strong desire to be able to practice and apply deep learning using Python.
- ▷ You downloaded, installed and started using Keras, perhaps for the first time, and started to get familiar with how to develop neural network models with the library.
- ▷ Slowly and steadily over the course of a number of lessons you learned how to use the various different features of the Keras library on neural network and deeper models for classical tabular, image and textual data.
- ▷ Building upon the recipes for common machine learning tasks you worked through your first machine learning problems with using deep learning models end-to-end using Python.
- ▷ Using standard templates, the recipes and experience you have gathered, you are now capable of working through new and different predictive modeling machine learning problems with deep learning on your own.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable skill of being able to work through machine learning problems with deep learning end-to-end using Python. This is a platform that is used by a majority of working data scientist professionals. The sky is the limit.

I want to take a moment and sincerely thank you for letting me help you start your deep learning journey with Python. I hope you keep learning and have fun as you continue to master machine learning.