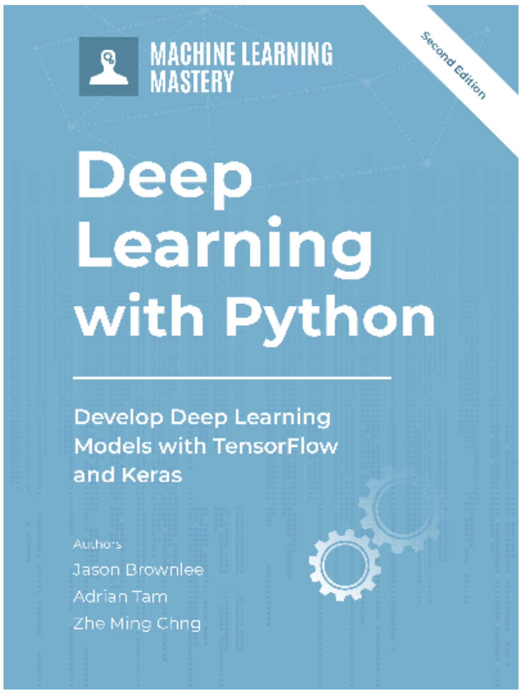
MACHINE LEARNING MASTERY

Second Edition

Deep Learning with Python

Develop Deep Learning Models with TensorFlow and Keras

Authors
Jason Brownlee
Adrian Tam
Zhe Ming Chng


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

Credits

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

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.

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:

```
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.

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:

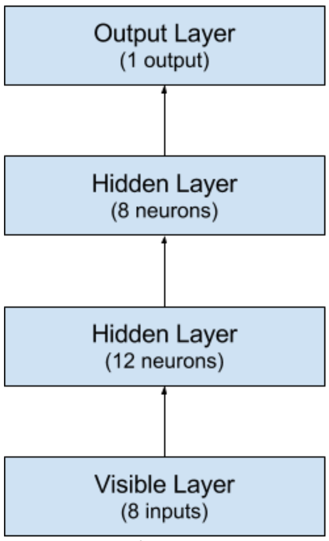


Figure 7.1: Model of a simple neural network

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:

```python
...
# 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:

```python
...
# 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.

```python
# 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]
```

```python
# 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

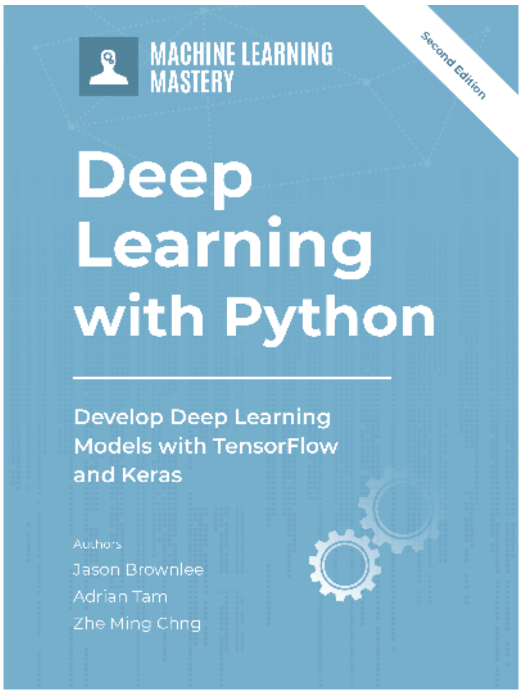
Keras official homepage.
    https://keras.io
Keras API Reference.
    https://keras.io/api/

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.



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