Beginning Application Development with TensorFlow and Keras

Learn to design, develop, train, and deploy TensorFlow and Keras models as real-world applications

Packt

www.packt.com

By Luis Capelo

Table of Contents

- Beginning Application Development with TensorFlow and Keras
    - Why subscribe?
    - PacktPub.com
- Contributors
    - About the author
    - About the reviewer
- Preface
    - What This Book Covers
    - What You Need for This Book
    - Who This Book is for
    - Conventions
    - Reader Feedback
    - Customer Support
    - Downloading the Example Code
    - Installation
        - Installing Visual Studio
        - Installing Python 3
        - Installing TensorFlow
        - Installing Keras
    - Errata
    - Piracy
    - Questions
- 1. Introduction to Neural Networks and Deep Learning
    - Lesson Objectives
    - What are Neural Networks?
        - Successful Applications
            - Why Do Neural Networks Work So Well?
                - Representation Learning
                - Function Approximation
            - Limitations of Deep Learning
                - Inherent Bias and Ethical Considerations
        - Common Components and Operations of Neural Networks
    - Configuring a Deep Learning Environment
        - Software Components for Deep Learning
            - Python 3
            - TensorFlow
            - Keras
            - TensorBoard
            - Jupyter Notebooks, Pandas, and NumPy
            - Activity 1 – Verifying Software Components
                - Exploring a Trained Neural Network
                    - MNIST Dataset
                        - Training a Neural Network with TensorFlow

- Training a Neural Network
- Testing Network Performance with Unseen Data
- Activity 2 – Exploring a Trained Neural Network
- Summary
- 2. Model Architecture
  - Lesson Objectives
  - Choosing the Right Model Architecture
    - Common Architectures
      - Convolutional Neural Networks
      - Recurrent Neural Networks
      - Generative Adversarial Networks
      - Deep Reinforcement Learning
    - Data Normalization
      - Z-score
      - Point-Relative Normalization
      - Maximum and Minimum Normalization
    - Structuring Your Problem
    - Activity 3 – Exploring the Bitcoin Dataset and Preparing Data for Model
  - Using Keras as a TensorFlow Interface
    - Model Components
    - Activity 4 – Creating a TensorFlow Model Using Keras
  - From Data Preparation to Modeling
    - Training a Neural Network
      - Reshaping Time-Series Data
    - Making Predictions
      - Overfitting
    - Activity 5 – Assembling a Deep Learning System
  - Summary
- 3. Model Evaluation and Optimization
  - Lesson Objectives
  - Model Evaluation
    - Problem Categories
    - Loss Functions, Accuracy, and Error Rates
      - Different Loss Functions, Same Architecture
      - Using TensorBoard
  - Implementing Model Evaluation Metrics
    - Evaluating the Bitcoin Model
      - Overfitting
      - Model Predictions
        - Interpreting Predictions
        - Activity 6 – Creating an Active Training Environment
      - Hyperparameter Optimization
        - Layers and Nodes - Adding More Layers
          - Adding More Nodes
          - Layers and Nodes - Implementation
          - Epochs
            - Epochs - Implementation
            - Activation Functions

Linear (Identity)
Hyperbolic Tangent (Tanh)
Rectified Linear Unit
Activation Functions - Implementation
Regularization Strategies
L2 Regularization
Dropout
Regularization Strategies – Implementation
Optimization Results
Activity 7 – Optimizing a Deep Learning Model
Summary
4. Productization
Lesson Objectives
Handling New Data
Separating Data and Model
Data Component
Model Component
Dealing with New Data
Re-Training an Old Model
Training a New Model
Activity 8 – Dealing with New Data
Deploying a Model as a Web Application
Application Architecture and Technologies
Deploying and Using Cryptonic
Activity 9 – Deploying a Deep Learning
Application
Summary
Index

Beginning Application Development with TensorFlow and Keras

Beginning Application Development with TensorFlow and Keras

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Acquisition Editor: Koushik Sen

Development Editor: Tanmayee Patil

Production Coordinator: Vishal Pawar, Samita Warang

First published: April 2018

Production reference: 1300418

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78953-729-1

www.packtpub.com

https://mapt.packtpub.com/

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal

development and advance your career. For more information, please visit https://mapt.packtpub.com/ website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <`service@packtpub.com`> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Luis Capelo is a Harvard-trained analyst and programmer who specializes in the design and development of data science products. He is based in the great New York City, USA.

He is the head of the Data Products team at Forbes, where they both investigate new techniques for optimizing article performance and create clever bots that help them distribute their content. Previously, he led a team of world-class scientists at the Flowminder Foundation, where we developed predictive models for assisting the humanitarian community. Prior to that, he worked for the United Nations as part of the Humanitarian Data Exchange team (founders of the Center for Humanitarian Data).

He is a native of Havana, Cuba, and the founder and owner of a small consultancy firm dedicated to supporting the nascent Cuban private sector.

About the reviewer

Manoj Pandey is a Python programmer and the founder and organizer of PyData Delhi. He works on research and development from time to time, and is currently working with RaRe Technologies on their incubator program for a computational linear algebra project. Prior to this, he has worked with Indian startups and small design/development agencies, and teaches Python/JavaScript to many on Codementor (@manojpandey). You can reach out to him at Twitter: onlyrealmvp.

Preface

TensorFlow is one of the most popular architectures used for machine learning and, more recently, deep learning. This book is your guide to deploy TensorFlow and Keras models into real-world applications.

The book begins with a dedicated blueprint for how to build an application that generates predictions. Each subsequent lesson tackles a particular type of model, such as neural networks, configuring a deep learning environment, using Keras and focuses on the three important questions of how the model works, how to improve our prediction accuracy in our example model, and how to measure and assess its performance using real-world applications.

In this book, you will learn how to create an application that generates predictions from deep learning. This learning journey begins by exploring the common components of a neural network and its essential performance. By end of the lesson you will be exploring a trained neural network created using TensorFlow. In the remaining lessons, you will learn to build a deep learning model with different components together and measuring their performance in prediction. Finally, we will be able to deploy a working web-application

By the end of this book, you will be equipped to create more accurate prediction by creating a completely new model, changing the core components of the application as you see fit.

What This Book Covers

Lesson 1, Introduction to Neural Networks and Deep Learning, helps you set up and configure deep learning environment and start looking at individual models and case studies. It also discusses neural networks and its idea along with their origins and explores their power.

Lesson 2, Model Architecture, shows how to predict Bitcoin prices using deep learning model.

Lesson 3, Model Evaluation and Optimization, shows on how to evaluate a neural network model. We will modify the network's hyperparameters to improve its performance.

Lesson 4, Productization explains how to productize a deep learning model and also provides an exercise of how to deploy a model as a web application.

What You Need for This Book

This book will require the following minimum hardware requirements:

- Processor: 1.8 GHz or higher
- Memory: 2 GB RAM
- Hard disk: 10 GB

Throughout this book, we will be using Python 3, TensorFlow, TensorBoard, and Keras. Please ensure you have the following installed on your machine:

- Code editor such as: Visual Studio Code (https://code.visualstudio.com/)
- Python 3.6
- TensorFlow 1.4 or higher on Windows
- Keras 2
- TensorBoard
- Jupyter Notebook
- Pandas
- NumPy
- Operating System: Windows (8 or higher), MacOS, or Linux (Ubuntu)

Who This Book is for

This book is designed for developers, analysts, and data scientists interested in developing applications using TensorFlow and Keras. You need to have programming knowledge. We also assume your familiarity with Python 3 and basic knowledge of web-applications. You also need to have a prior understanding and working knowledge of linear algebra, probability, and statistics.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `\` class provides static methods to generate an instance of itself, such as `()`."

A block of code is set as follows:

```
tf.nn.max_pool(
activation,
ksize=[1, 2, 2, 1],
strides=[1, 2, 2, 1],
padding="SAME")
```

Any command-line input or output is written as follows:

```
$ python3
lesson_1/activity_1/test_stack.py
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the Next button moves you to the next screen."

NOTE

Warnings or important notes appear in a box like this.

TIP

Tips and tricks appear like this.

Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <`feedback@packtpub.com`>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the Example Code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads** & **Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/TrainingByPackt/Beginning-Application-Developmentwith-TensorFlow-and-Keras. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Installation

Before you start with this course, we'll install Visual Studio Code, Python 3, TensorFlow, and Keras. The steps for installation are as follows:

Installing Visual Studio

1. Visit https://code.visualstudio.com/ in your browser.
2. Click on Download in the top-right corner of the home page.
3. Next, select Windows.
4. Follow the steps in the installer and that's it! Your Visual Studio Code is ready.

Installing Python 3

1. Go to https://www.python.org/downloads/.
2. Click on the Download Python 3.6.4 option to dowload the setup.
3. Follow the steps in the installer and that's it! Your Python is ready.

Installing TensorFlow

Download and install TensorFlow by following the instructions on this website:https://www.tensorflow.org/install/install_windows.

Installing Keras

Download and install Keras by following the instructions on this website: https://keras.io/#installation.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the **Errata** section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <`copyright@packtpub.com`> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at <`questions@packtpub.com`>, and we will do our best to address the problem.

Chapter 1. Introduction to Neural Networks and Deep Learning

In this lesson, we will cover the basics of neural networks and how to set up a deep learning programming environment. We will also explore the common components of a neural network and its essential operations. We will conclude this lesson by exploring a trained neural network created using TensorFlow.

This lesson is about understanding what neural networks can do. We will not cover mathematical concepts underlying deep learning algorithms, but will instead describe the essential pieces that make a deep learning system. We will also look at examples where neural networks have been used to solve real-world problems.

This lesson will give you a practical intuition on how to engineer systems that use neural networks to solve problems—including how to determine if a given problem can be solved at all with such algorithms. At its core, this lesson challenges you to think about your problem as a mathematical representation of ideas. By the end of this lesson, you will be able to think about a problem as a collection of these representations and then start to recognize how these representations may be learned by deep learning algorithms.

Lesson Objectives

By the end of this lesson, you will be able to:

- Cover the basics of neural networks
- Set up a deep learning programming environment
- Explore the common components of a neural network and its essential operations
- Conclude this lesson by exploring a trained neural network created using TensorFlow

What are Neural Networks?

Neural networks—also known as **Artificial Neural Networks**—were first proposed in the 40s by MIT professors Warren McCullough and Walter Pitts.

NOTE

For more information refer, *Explained: Neural networks*. MIT News Office, April 14, 2017. Available at: http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

Inspired by advancements in neuroscience, they proposed to create a computer system that reproduced how the brain works (human or otherwise). At its core was the idea of a computer system that worked as an interconnected network. That is, a system that has many simple components. These components both interpret data and influence each other on how to interpret data. This same core idea remains today.

Deep learning is largely considered the contemporary study of neural networks. Think of it as a current name given to neural networks. The main difference is that the neural networks used in deep learning are typically far greater in size—that is, they have many more nodes and layers—than earlier neural networks. Deep learning algorithms and applications typically require resources to achieve success, hence the use of the word *deep* to emphasize its size and the large number of interconnected components.

Successful Applications

Neural networks have been under research since their inception in the 40s in one form or another. It is only recently, however, that deep learning systems have been successfully used in large-scale industry applications.

Contemporary proponents of neural networks have demonstrated great success in speech recognition, language translation, image classification, and other fields. Its current prominence is backed by a significant increase in available computing power and the emergence of **Graphic Processing Units** (**GPUs**) and **Tensor Processing Units** (**TPUs**)—which are able to perform many more simultaneous mathematical operations than regular CPUs, as well as a much greater availability of data.

Power consumption of different AlphaGo algorithms. AlphaGo is an initiative by DeepMind to develop a series of algorithms to beat the game Go. It is considered a prime example of the power of deep learning. TPUs are a chipset developed by Google for the use in deep learning programs.

The graphic depicts the number of GPUs and TPUs used to train different versions of the AlphaGo algorithm. Source: https://deepmind.com/blog/alphago-zero-learning-scratch/.

NOTE

In this book, we will not be using GPUs to fulfil our activities. GPUs are not required to work with neural networks. In a number of simple examples—like the ones provided in this book—all computations can be performed using a simple laptop's CPU. However, when dealing with very large datasets, GPUs can be of great help given that the long time to train a neural network would be unpractical.

Here are a few examples of fields in which neural networks have had great impact:

- **Translating text**: In 2017, Google announced that it was releasing a new algorithm for its translation service called **Transformer**. The algorithm consisted of a recurrent neural network (LSTM) that is trained used bilingual text. Google showed that its algorithm had gained notable accuracy when comparing to industry standards (BLEU) and was also computationally efficient. At the time of writing, Transformer is reportedly used by Google Translate as its main translation algorithm.

  NOTE

  Google Research Blog. *Transformer: A Novel Neural Network Architecture for Language Understanding*. August 31, 2017. Available at: https://research.googleblog.com/2017/08/transformer-novel-neuralnetwork.html.

- **Self-driving vehicles**: Uber, NVIDIA, and Waymo are believed to be using deep learning models to control different vehicle functions that control driving. Each company is researching a number of possibilities, including training the network using humans, simulating vehicles driving in virtual environments, and even creating a small city-like environment in which vehicles can be trained based on expected and unexpected events.

  NOTE

  - Alexis C. Madrigal: *Inside Waymo's Secret World for Training Self-Driving Cars*. The Atlantic. August 23, 2017. Available at: https://www.theatlantic.com/technology/archive/2017/08/inside-waymos-secret-testing-and-simulation-facilities/537648/">lities/537648/.
  - NVIDIA: *End-to-End Deep Learning for Self-Driving Cars*. August 17, 2016. Available at: https://devblogs.nvidia.com/parallelforall/deeplearning-self-driving-cars/.
  - Dave Gershgorn: *Uber's new AI team is looking for the shortest route to self-

driving cars. Quartz. December 5, 2016. Available at: https://qz.com/853236/ubers-new-ai-team-is-looking-for-the-shortest-route-to-self-driving-cars/.

- **Image recognition**: Facebook and Google use deep learning models to identify entities in images and automatically tag these entities as persons from a set of contacts. In both cases, the networks are trained with previously tagged images as well as with images from the target friend or contact. Both companies report that the models are able to suggest a friend or contact with a high level of accuracy in most cases.

While there are many more examples in other industries, the application of deep learning models is still in its infancy. Many more successful applications are yet to come, including the ones that you create.

WHY DO NEURAL NETWORKS WORK SO WELL?

Why are neural networks so powerful? Neural networks are powerful because they can be used to predict any given function with reasonable approximation. If one is able to represent a problem as a mathematical function and also has data that represents that function correctly, then a deep learning model can, in principle—and given enough resources—be able to approximate that function. This is typically called the *universality principle of neural networks*.

NOTE

For more information refer, Michael Nielsen: *Neural Networks and Deep Learning: A visual proof that neural nets can compute any function*. Available at: http://neuralnetworksanddeeplearning.com/chap4.html.

We will not be exploring mathematical proofs of the universality principle in this book. However, two characteristics of neural networks should give you the right intuition on how to understand that principle: representation learning and function approximation.

NOTE

For more information refer, Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. *A Brief Survey of Deep Reinforcement Learning*. arXiv. September 28, 2017. Available at: https://www.arxiv-vanity.com/papers/1708.05866/.

Representation Learning

The data used to train a neural network contains representations (also known as *features*) that explain the problem you are trying to solve. For instance, if one is interested in recognizing faces from images, the color values of each pixel from a set of images that contain faces will be used as a starting point. The model will then continuously learn higher-level representations by combining pixels together as it goes through its training process.

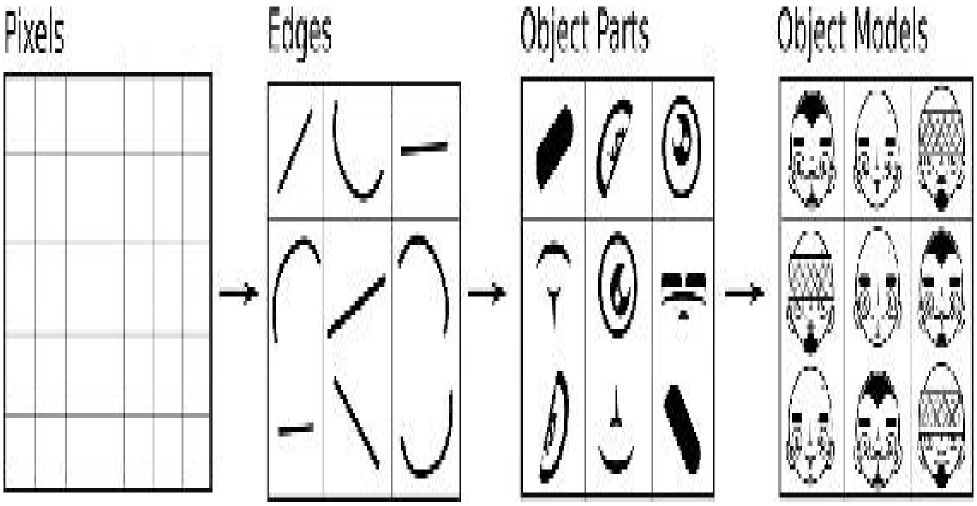


Figure 1: Series of higher-level representations that begin on input data. Derivate image based on original image from: Yann LeCun, Yoshua Bengio & Geoffrey Hinton. "Deep Learning". Nature 521, 436–444 (28 May 2015) doi:10.1038/nature14539

In formal words, neural networks are computation graphs in which each step computes higher abstraction representations from input data.

Each one of these steps represents a progression into a different abstraction layer. Data progresses through these layers, building continuously higher-level representations. The process finishes with the highest representation possible: the one the model is trying to predict.

Function Approximation

When neural networks learn new representations of data, they do so by combining weights and biases with neurons from different layers. They adjust the weights of these connections every time a training cycle takes place using a mathematical technique called backpropagation. The weights and biases improve at each round, up to the point that an optimum is achieved. This means that a neural network can measure how wrong it is on every training cycle, adjust the weights and biases of each neuron, and try again. If it determines that a certain modification produces better results than the previous round, it will invest in that modification until an optimal solution is achieved.

In a nutshell, that procedure is the reason why neural networks can approximate functions. However, there are many reasons why a neural network may not be able to predict a function with perfection, chief among them being that:

- Many functions contain stochastic properties (that is, random properties)
- There may be overfitting to peculiarities from the training data
- There may be a lack of training data

In many practical applications, simple neural networks are able to approximate a function with reasonable precision. These sorts of applications will be our focus throughout this book.

LIMITATIONS OF DEEP LEARNING

Deep learning techniques are best suited to problems that can be defined with formal mathematical rules (that is, as data representations). If a problem is hard to define this way, then it is likely that deep learning will not provide a useful solution. Moreover, if the data available for a given problem is either biased or only contains partial representations of the underlying functions that generate that problem, then deep learning techniques will only be able to reproduce the problem and not learn to solve it.

Remember that deep learning algorithms are learning different representations of data to approximate a given function. If data does not represent a function appropriately, it is likely that a function will be incorrectly represented by a neural network. Consider the following analogy: you are trying to predict the national prices of gasoline (that is, fuel) and create a deep learning model. You use your credit card statement with your daily expenses on gasoline as an input data for that model. The model may eventually learn the patterns of your gasoline consumption, but it will likely misrepresent price fluctuations of gasoline caused by other factors only represented weekly in your data such as government policies, market competition, international politics, and so on. The model will ultimately yield incorrect results when used in production.

To avoid this problem, make sure that the data used to train a model represents the problem the model is trying to address as accurately as possible.

> **NOTE**
>
> For an in-depth discussion of this topic, refer to François Chollet's upcoming book *Deep Learning with Python*. François is the creator of Keras, a Python library used in this book. The chapter, *The limitations of deep learning,* is particularly important for understanding this topic. The working version of that book is available at: https://blog.keras.io/the-limitations-of-deep-learning.html.

Inherent Bias and Ethical Considerations

Researchers have suggested that the use of the deep learning model without considering the inherent bias in the training data can lead not only to poor performing solutions, but also to ethical complications.

For instance, in late 2016, researchers from the Shanghai Jiao Tong University in China created a neural network which correctly classified criminals using only pictures from their faces. The researchers used 1,856 images of Chinese men in which half had been convicted.

NOTE

Their model identified inmates with 89.5 percent accuracy. (https://blog.keras.io/the-limitations-of-deep-learning.htmltations-of-deeplearning.html).

MIT Technology Review. Neural Network Learns to Identify Criminals by Their Faces. November 22, 2016. Available at: https://www.technologyreview.com/s/602955/neuralnetwork-learns-to-identify-criminals-by-their-faces/.

The paper resulted in great furor within the scientific community and popular media. One key issue with the proposed solution is that it fails to properly recognize the bias inherent in the input data. Namely, the data used in this study came from two different sources: one for criminals and one for non-criminals. Some researchers suggest that their algorithm identifies patterns associated with the different data sources used in the study instead of identifying relevant patterns from people's faces. While there are technical considerations one can make about the reliability of the model, the key criticism is on ethical grounds: one ought to clearly recognize the inherent bias in input data used by deep learning algorithms and consider how its application will have an impact on people's lives.

NOTE

Timothy Revell. *Concerns as face recognition tech used to 'identify' criminals*. New Scientist. December 1, 2016. Available at: https://www.newscientist.com/article/2114900-concerns-as-face-recognition-tech-used-to-identify-criminals/.

For understanding more about the topic of ethics in learning algorithms (including deep learning), refer to the work done by the AI Now Institute (https://ainowinstitute.org/), an organization created for the understanding of the social implications of intelligent systems.

Common Components and Operations of Neural Networks

Neural networks have two key components: layers and nodes.

Nodes are responsible for specific operations, and layers are groups of nodes used to differentiate different stages of the system. Typically, neural networks have the following three categories of layers:

- **Input**: Where the input data is received and first interpreted
- **Hidden**: Where computations take place, modifying data as it goes through
- **Output**: Where an output is assembled and evaluated

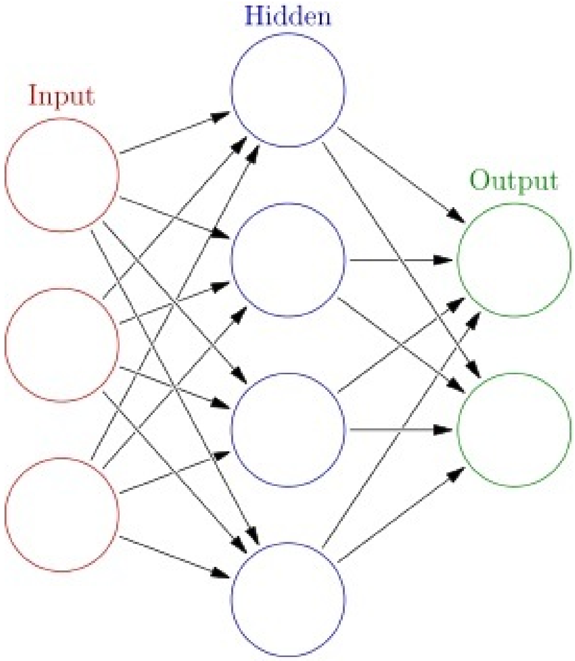


Figure 2: Illustration of the most common layers in a neural network. By Glosser.ca - Own work, Derivative of File: Artificial neural network.svg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=24913461

Hidden layers are the most important layers in neural networks. They are referred to as *hidden* because the representations generated in them are not available in the data, but are learned from it. It is within these layers where the main computations take place in neural networks.

Nodes are where data is represented in the network. There are two values associated with nodes: biases and weights. Both of these values affect how data is represented by the nodes and passed on to other nodes.

When a network *learns,* it effectively adjusts these values to satisfy an optimization function.

Most of the work in neural networks happens in the hidden layers. Unfortunately, there isn't a clear rule for determining how many layers or nodes a network should have. When implementing a neural network, one will probably spend time experimenting with different combinations of layers and nodes. It is advised to always start with a single layer and also with a number of nodes that reflect the number of features the input data has (that is, how many *columns* are available in a dataset). One will then continue to add layers and nodes until satisfactory performance is achieved—or whenever the network starts overfitting to the training data.

Contemporary neural network practice is generally restricted to the experimentation with the number of nodes and layers (for example, how deep the network is), and the kinds of operations performed at each layer. There are many successful instances in which neural networks outperformed other algorithms simply by adjusting these parameters.

As an intuition, think about data entering a neural network system via the input layer, then moving through the network from node to node. The path that data takes will depend on how interconnected the nodes are, the weights and the biases of each node, the kind of operations that are performed in each layer, and the state of data at the end of such operations. Neural networks often require many "runs" (or epochs) in order to keep tuning the weights and biases of nodes, meaning that data flows over the different layers of the graph multiple times.

This section offered you an overview of neural networks and deep learning. Additionally, we discussed a starter's intuition to understand the following key concepts:

- Neural networks can, in principle, approximate most functions, given that it has enough resources and data.
- Layers and nodes are the most important structural components of neural networks. One typically spends a lot of time altering those components to find a working architecture.
- Weights and biases are the key properties that a network "learns" during its training process.

Those concepts will prove useful in our next section as we explore a real-world trained neural network and make modifications to train our own.

Configuring a Deep Learning Environment

Before we finish this lesson, we want you to interact with a real neural network. We will start by covering the main software components used throughout this book and make sure that they are properly installed. We will then explore a pre-trained neural network and explore a few of the components and operations discussed earlier in the *What are Neural Networks?* section.

Software Components for Deep Learning

We'll use the following software components for deep learning:

PYTHON 3

We will be using Python 3 in this book. Python is a general-purpose programming language which is very popular with the scientific community—hence its adoption in deep learning. Python 2 is not supported in this book but can be used to train neural networks instead of Python 3. Even if you chose to implement your solutions in Python 2, consider moving to Python 3 as its modern feature set is far more robust than that of its predecessor.

TENSORFLOW

TensorFlow is a library used for performing mathematical operations in the form of graphs. TensorFlow was originally developed by Google and today it is an open-source project with many contributors. It has been designed with neural networks in mind and is among the most popular choices when creating deep learning algorithms.

TensorFlow is also well-known for its production components. It comes with TensorFlow Serving (https://github.com/tensorflow/serving), a high-performance system for serving deep learning models. Also, trained TensorFlow models can be consumed in other high-performance programming languages such as Java, Go, and C. This means that one can deploy these models in anything from a micro-computer (that is, a RaspberryPi) to an Android device.

KERAS

In order to interact efficiently with TensorFlow, we will be using Keras (https://keras.io/), a Python package with a high-level API for developing neural networks. While TensorFlow focuses on components that interact with each other in a computational graph, Keras focuses specifically on neural networks. Keras uses TensorFlow as its backend engine and makes developing such applications much easier.

As of November 2017 (TensorFlow version 1.4), Keras is distributed as part of TensorFlow. It is available under the `tf.keras` namespace. If you have TensorFlow 1.4 or higher installed, you already have Keras available in your system.

TENSORBOARD

TensorBoard is a data visualization suite for exploring TensorFlow models and is natively integrated with TensorFlow. TensorBoard works by consuming checkpoint and summary files created by TensorFlow as it trains a neural network. Those can be explored either in near real-time (with a 30 second delay) or after the network has finished training. TensorBoard makes the process of experimenting and exploring a neural network much easier—plus it's quite exciting to follow the training of your network!

JUPYTER NOTEBOOKS, PANDAS, AND NUMPY

When working to create deep learning models with Python, it is common to start working interactively, slowly developing a model that eventually turns into more structured software. Three Python packages are used frequently during that process: Jupyter Notebooks, Pandas, and NumPy:

- Jupyter Notebooks create interactive Python sessions that use a web browser as its interface
- Pandas is a package for data manipulation and analysis
- NumPy is frequently used for shaping data and performing numerical

computations

These packages are used occasionally throughout this book. They typically do not form part of a production system, but are often used when exploring data and starting to build a model. We focus on the other tools in much more detail.

NOTE

The book *Learning Pandas* by Michael Heydt (June 2017, Packt Publishing) and *Learning Jupyter* by Dan Toomey (November 2016, Packt Publishing) offer a comprehensive guide on how to use these technologies. These books are good references for continuing to learn more.

Component	Description	Minimum Version
Python	General-purpose programming language. Popular language used in the development of deep learning applications.	3.6
TensorFlow	Open-source graph computation Python package typically used for developing deep learning systems.	1.4
Keras	Python package that provides a high-level interface to TensorFlow.	2.0.8-tf (distributed with TensorFlow)
Tensor	Browser-based software for visualizing neural	0.4.0

TensorBoard	Browser-based software for visualizing neural network statistics.	0.4.0
Jupyter Notebook	Browser-based software for working interactively with Python sessions.	5.2.1
Pandas	Python package for analyzing and manipulating data.	0.21.0
NumPy	Python package for high-performance numerical computations.	1.13.3

Table 1: Software components necessary for creating a deep learning environment

ACTIVITY 1 – VERIFYING SOFTWARE COMPONENTS

Before we explore a trained neural network, let's verify that all the software components that we need are available. We have included a script that verifies these components work. Let's take a moment to run the script and deal with any eventual problems we may find.

We will now be testing if the software components required for this book are available in your working environment. First, we suggest the creation of a Python virtual environment using Python's native module `venv`. Virtual environments are used for managing project dependencies. We advise each project you create to have its own virtual environments. Let's create one now.

> **NOTE**
>
> If you are more comfortable with conda environments, feel free to use those instead.

1. A Python virtual environment can be created by using the following command:

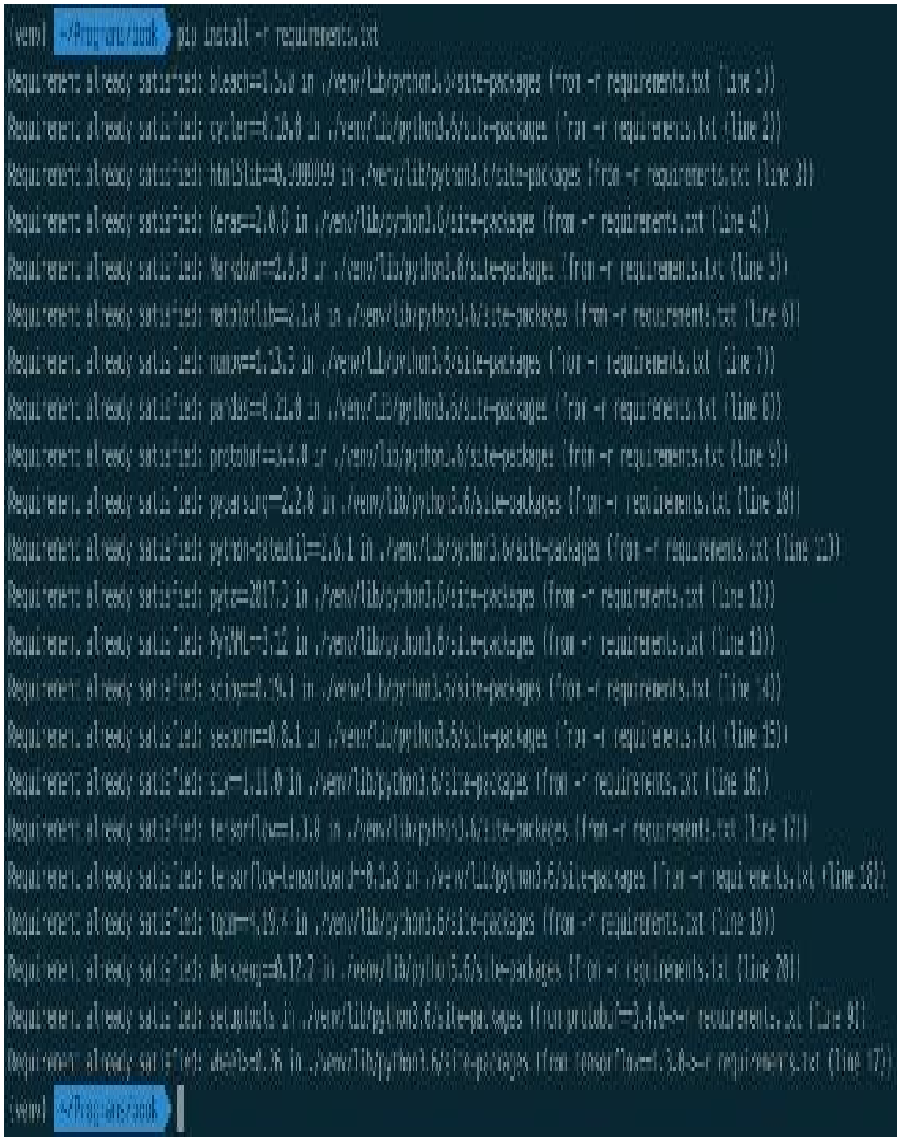
   ```
   $ python3 -m venv venv
   $ source venv/bin/activate
   ```

2. The latter command will append the string (`venv`) to the beginning of your command line. Use the following command to deactivate your virtual environment:

   ```
   $ deactivate
   ```

   > **NOTE**
   >
   > Make sure to always activate your Python virtual environment when working on a project.

3. After activating your virtual environment, make sure that the right components are installed by executing `pip` over the file `requirements.txt`. This will attempt to install the models used in this book in that virtual environment. It will do nothing if they are already available:



```
(venv) ~/Programs/book pip install -r requirements.txt
Requirement already satisfied: bleach==1.5.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 1))
Requirement already satisfied: cycler==0.10.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 2))
Requirement already satisfied: html5lib==0.9999999 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 3))
Requirement already satisfied: Keras==2.0.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 4))
Requirement already satisfied: Markdown==2.6.9 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 5))
Requirement already satisfied: matplotlib==2.1.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 6))
Requirement already satisfied: numpy==1.13.3 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 7))
Requirement already satisfied: pandas==0.21.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 8))
Requirement already satisfied: protobuf==3.4.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 9))
Requirement already satisfied: pyparsing==2.2.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 10))
Requirement already satisfied: python-dateutil==2.6.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 11))
Requirement already satisfied: pytz==2017.3 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 12))
Requirement already satisfied: PyYAML==3.12 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 13))
Requirement already satisfied: scipy==0.19.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 14))
Requirement already satisfied: seaborn==0.8.1 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 15))
Requirement already satisfied: six==1.11.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 16))
Requirement already satisfied: tensorflow==1.3.0 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 17))
Requirement already satisfied: tensorflow-tensorboard==0.1.8 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 18))
Requirement already satisfied: tqdm==4.19.4 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 19))
Requirement already satisfied: Werkzeug==0.12.2 in ./venv/lib/python3.6/site-packages (from -r requirements.txt (line 20))
Requirement already satisfied: setuptools in ./venv/lib/python3.6/site-packages (from protobuf==3.4.0->-r requirements.txt (line 9))
Requirement already satisfied: wheel>=0.26 in ./venv/lib/python3.6/site-packages (from tensorflow==1.3.0->-r requirements.txt (line 17))
(venv) ~/Programs/book
```

Figure 3: Image of a terminal running pip to install dependencies from requirements.txt

Install dependencies by running the following command:

```
$ pip install –r requirements.txt
```

This will install all required dependencies for your system. If they are already installed, this command should simply inform you.

These dependencies are essential for working with all code activities in this book.

As a final step on this activity, let's execute the script `test_stack.py`. That script formally verifies that all the required packages for this book are installed and available in your system.

4. Students, run the script `lesson_1/activity_1/test_stack.py` to check if the dependencies Python 3, TensorFlow, and Keras are available. Use the following command:

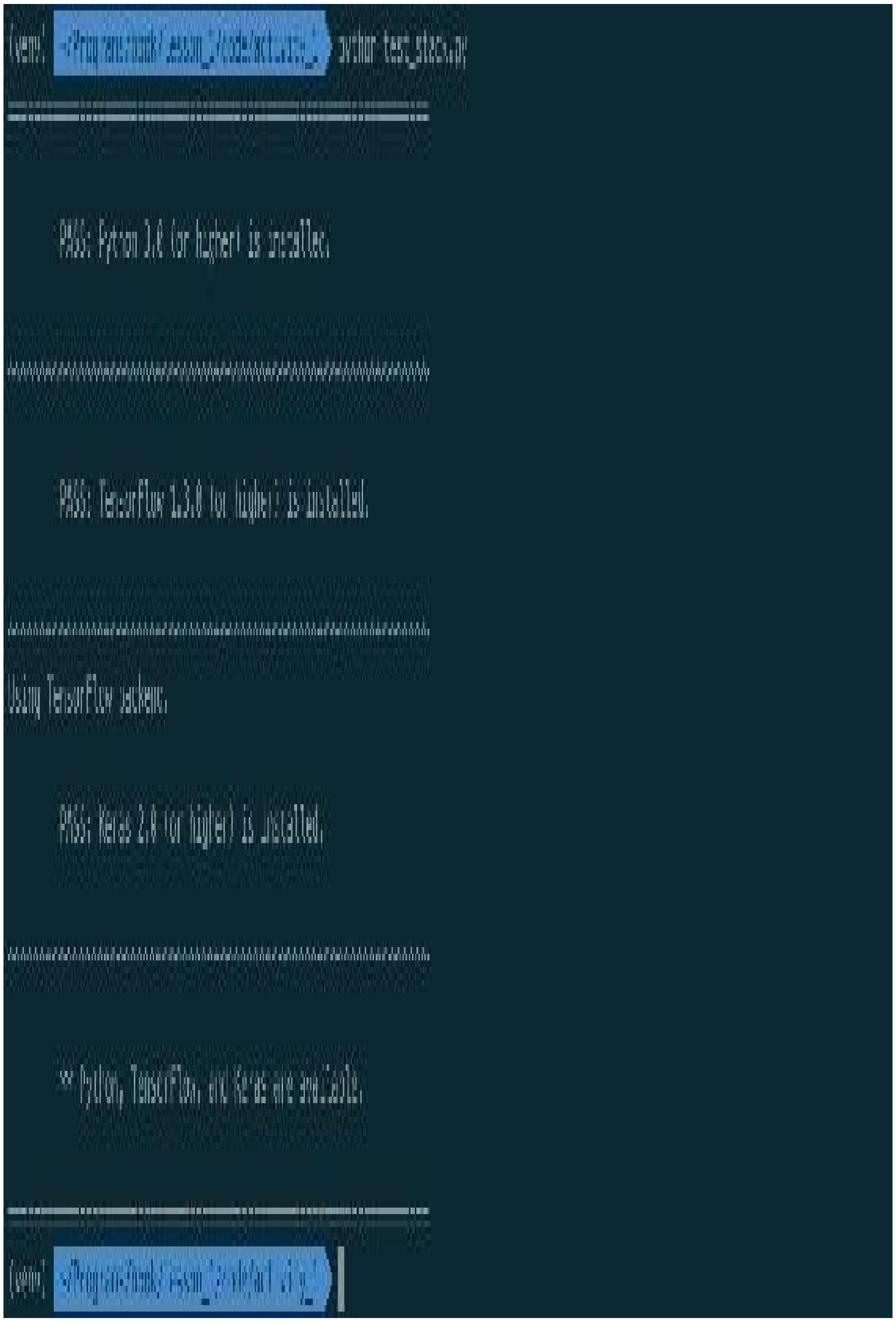
   ```
   $ python3
   lesson_1/activity_1/test_stack.py
   ```

   The script returns helpful messages stating what is installed and what needs to be installed.

5. Run the following script command in your terminal:

   ```
   $ tensorboard --help
   ```

   You should see a help message that explains what each command does. If you do not see that message – or see an error message instead – please ask for assistance from your instructor:



```
(venv) [illegible] python test_stack.py
==============================================================

    PASS: Python 3.6 (or higher) is installed.

--------------------------------------------------------------

    PASS: TensorFlow 1.3.0 (or higher) is installed.

--------------------------------------------------------------
Using TensorFlow backend.

    PASS: Keras 2.0 (or higher) is installed.

--------------------------------------------------------------

    ** Python, TensorFlow, and Keras are available.

==============================================================
(venv) [illegible]
```

Figure 4: Image of a terminal running python3 test_stack.py. The script returns messages informing that all dependencies are installed correctly.

NOTE

If a similar message to the following appears, there is no need to worry:

```
RuntimeWarning: compiletime version 3.5 of module
'tensorflow.python.framework.fast_tensor_util'
```

```
'tensorflow.python.framework.fast_tensor_util'
does not match runtime version 3.6
return f(*args, **kwds)
```

That message appears if you are running Python 3.6 and the distributed TensorFlow wheel was compiled under a different version (in this case, 3.5). You can safely ignore that message.

Once we have verified that Python 3, TensorFlow, Keras, TensorBoard, and the packages outlined in `requirements.txt` have been installed, we can continue to a demo on how to train a neural network and then go on to explore a trained network using these same tools.

NOTE

For the reference solution, use the `Code/Lesson-1/activity_1` folder.

Exploring a Trained Neural Network

In this section, we explore a trained neural network. We do that to understand how a neural network solves a real-world problem (predict handwritten digits) and also to get familiar with the TensorFlow API. When exploring that neural network, we will recognize many components introduced in previous sections such as nodes and layers, but we will also see many that we don't recognize (such as activation functions)—we will explore those in further sections. We will then walk through an exercise on how that neural network was trained and then train that same network on our own.

The network that we will be exploring has been trained to recognize numerical digits (integers) using images of handwritten digits. It uses the MNIST dataset (http://yann.lecun.com/exdb/mnist/), a classic dataset frequently used for exploring pattern recognition tasks.

MNIST Dataset

The **Modified National Institute of Standards and Technology** (**MNIST**) dataset contains a training set of 60,000 images and a test set of 10,000 images. Each image contains a single handwritten number. This dataset—which is a derivate from one created by the US Government—was originally used to test different approaches to the problem of recognizing handwritten text by computer systems. Being able to do that was important for the purpose of increasing the performance of postal services, taxation systems, and government services. The MNIST dataset is considered too naïve for contemporary methods. Different and more recent datasets are used in contemporary research (for example, CIFAR). However, the MNIST dataset is still very useful for understanding how neural networks work because known models can achieve a high level of accuracy with great efficiency.

> **NOTE**
>
> The CIFAR dataset is a machine learning dataset that contains images organized in different classes. Different than the MNIST dataset, the CIFAR dataset contains classes in many different areas such as animals, activities, and objects. The CIFAR dataset is available at: https://www.cs.toronto.edu/~kriz/cifar.html.

Figure 5: Excerpt from the training set of the MNIST dataset. Each image is a separate 20x20 pixels image with a single handwritten digit. The original dataset is available at: http://yann.lecun.com/exdb/mnist/.

Training a Neural Network with TensorFlow

Now, let's train a neural network to recognize new digits using the MNIST dataset.

We will be implementing a special-purpose neural network called "Convolutional Neural Network" to solve this problem (we will discuss those in more detail in further sections). Our network contains three hidden layers: two fully connected layers and a convolutional layer. The **convolutional layer** is defined by the following TensorFlow snippet of Python code:

```
W = tf.Variable(
      tf.truncated_normal([5, 5, size_in,
size_out],
      stddev=0.1),
      name="Weights")

   B = tf.Variable(tf.constant(0.1, shape=
[size_out]), name="Biases")

   convolution = tf.nn.conv2d(input, W,
strides=[1, 1, 1, 1], padding="SAME")
   activation = tf.nn.relu(convolution +
B)

  tf.nn.max_pool(
  activation,
  activation,
  ksize=[1, 2, 2, 1],
  strides=[1, 2, 2, 1],
  padding="SAME")
```

NOTE

Use the `mnist.py` file for your reference at `Code/Lesson-1/activity_2/`. Follow along by opening the script in your code editor.

We execute that snippet of code only once during the training of our network.

The variables `W` and `B` stand for weights and biases. These are the values used by the nodes within the hidden layers to alter the network's interpretation of the data as it passes through the network. Do not worry about the other variables for now.

The **fully connected layers** are defined by the following snippet of Python code:

```
W = tf.Variable(
     tf.truncated_normal([size_in,
size_out], stddev=0.1),
          name="Weights")

 B = tf.Variable(tf.constant(0.1, shape=
[size_out]), name="Biases")
          activation = tf.matmul(input, W)
+ B
```

NOTE

Use the `mnist.py` file for your reference at `Code/Lesson-1/activity_2/`. Follow along by opening the script in your code editor.

Here, we also have the two TensorFlow variables `W` and

B. Notice how simple the initialization of these variables is: W is initialized as a random value from a pruned Gaussian distribution (pruned with size_in and size_out) with a standard deviation of 0.1, and B (the bias term) is initialized as 0.1, a constant. Both these values will continuously change during each run. That snippet is executed twice, yielding two fully connected networks—one passing data to the other.

Those 11 lines of Python code represent our complete neural network. We will go into a lot more detail in *Lesson 2*, *Model Architecture* about each one of those components using Keras. For now, focus on understanding that the network is altering the values of W and B in each layer on every run and how these snippets form different layers. These 11 lines of Python are the culmination of dozens of years of neural network research.

Let's now train that network to evaluate how it performs in the MNIST dataset.

Training a Neural Network

Follow the following steps to set up this exercise:

1. Open two terminal instances.
2. In both of them, navigate to the directory lesson_1/exercise_a.
3. In both of them, make sure that your Python 3 virtual environment is active and that the requirements outlined in requirements.txt are installed.
4. In one of them, start a TensorBoard server with the following command:

```
$ tensorboard --logdir=mnist_example/
$ tensorboard --logdir=mnist_example/
```

5. In the other, run the `train_mnist.py` script from within that directory.
6. Open your browser in the TensorBoard URL provided when you start the server.

In the terminal that you ran the script `train_mnist.py,` you will see a progress bar with the epochs of the model. When you open the browser page, you will see a couple of graphs. Click on the one that reads **Accuracy**, enlarge it and let the page refresh (or click on the **refresh** button). You will see the model gaining accuracy as epochs go by.

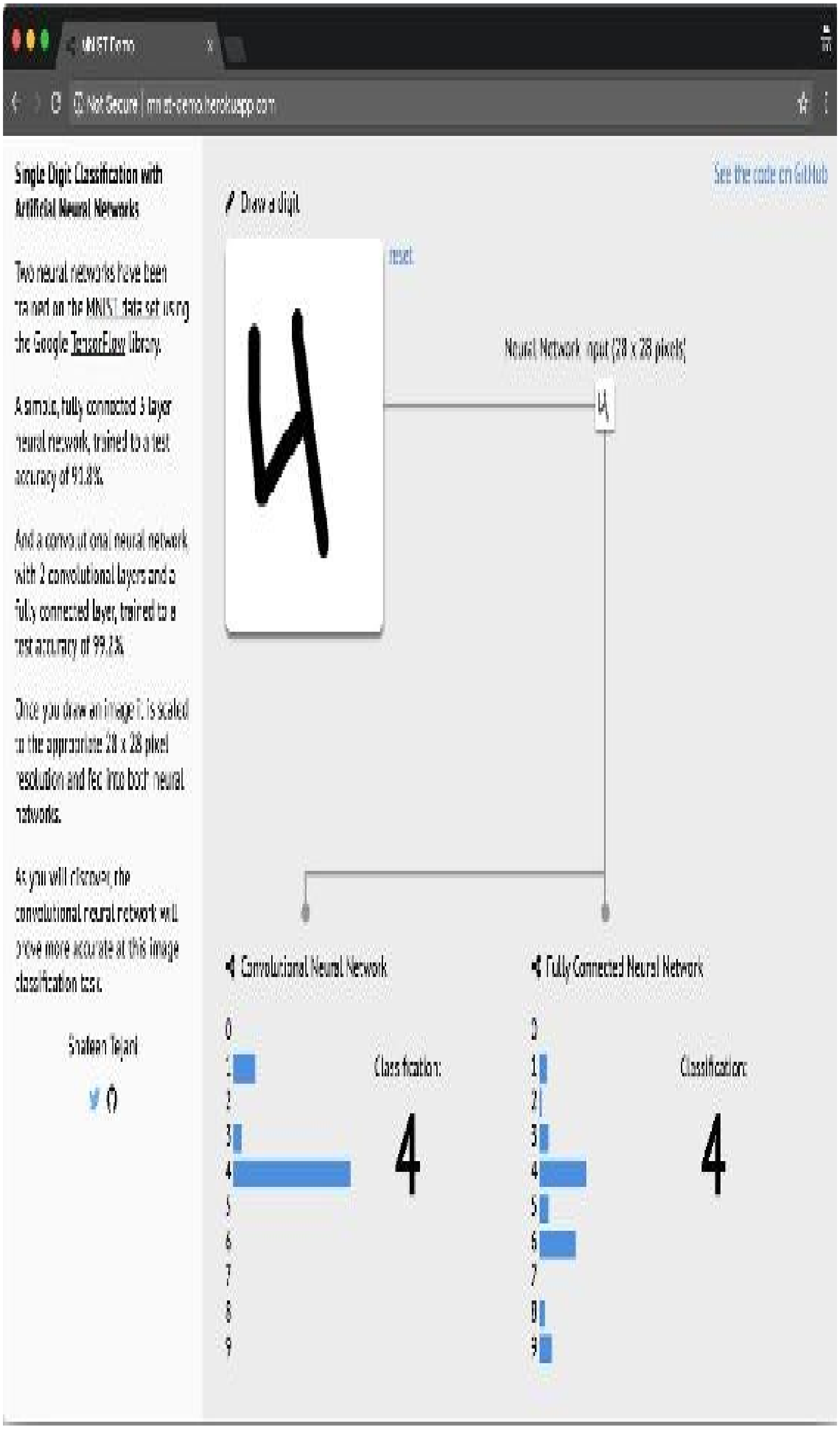
Use that moment to explain the power of neural networks in reaching a high level of accuracy very early in the training process.

We can see that in about 200 epochs (or steps), the network surpassed 90 percent accuracy. That is, the network is getting 90 percent of the digits in the test set correctly. The network continues to gain accuracy as it trains up to the 2,000th step, reaching a 97 percent accuracy at the end of that period.

Now, let's also test how well those networks perform with unseen data. We will use an open-source web application created by Shafeen Tejani to explore if a trained network correctly predicts handwritten digits that we create.

Testing Network Performance with Unseen Data

Visit the website http://mnist-demo.herokuapp.com/">mo.herokuapp.com/ in your browser and draw a number between **0** and **9** in the designated white box:



MNIST Demo
Not Secure | mnist-demo.herokuapp.com
See the code on GitHub
Draw a digit
reset
Neural Network input (28 x 28 pixels)
Convolutional Neural Network
Fully Connected Neural Network
0
1
2
3
4
5
6
7
8
9
Classification:
4
Classification:
4

Single Digit Classification with Artificial Neural Networks

Two neural networks have been trained on the MNIST data set using the Google TensorFlow library.

A simple, fully connected 3 layer neural network, trained to a test accuracy of 91.8%.

And a convolutional neural network with 2 convolutional layers and a fully connected layer, trained to a test accuracy of 99.2%.

Once you draw an image it is scaled to the appropriate 28 x 28 pixel resolution and fed into both neural networks.

As you will discover, the convolutional neural network will prove more accurate at this image classification task.

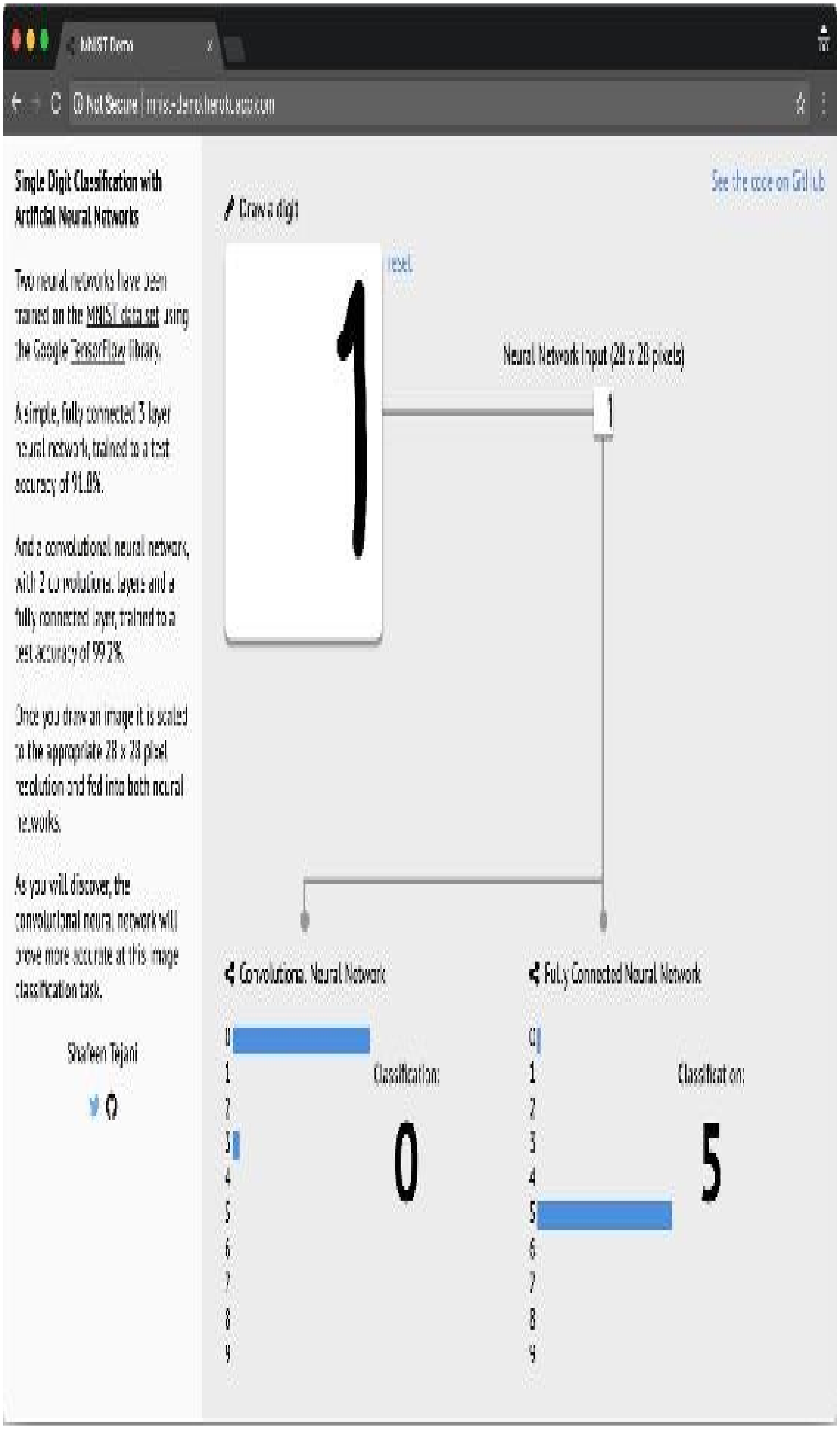
Shaleen Tejani

Figure 6: Web application in which we can manually draw digits and test the accuracy of two trained netwo

NOTE

Source: https://github.com/ShafeenTejani/mnist-demo

In the application, you can see the results of two neural networks. The one that we have trained is on the left (called CNN). Does it classify all your handwritten digits correctly? Try drawing numbers at the edge of the designated area. For instance, try drawing the number **1** close to the right edge of that area:



MNIST Demo
Not Secure | mnist-demo.herokuapp.com
See the code on GitHub
Draw a digit
reset
Neural Network Input (28 x 28 pixels)
Convolutional Neural Network
0
1
2
3
4
5
6
7
8
9
Classification:
0
Fully Connected Neural Network
0
1
2
3
4
5
6
7
8
9
Classification:
5

Single Digit Classification with Artificial Neural Networks

Two neural networks have been trained on the MNIST data set using the Google TensorFlow library.

A simple, fully connected 3 layer neural network, trained to a test accuracy of 91.8%.

And a convolutional neural network, with 2 convolutional layers and a fully connected layer, trained to a test accuracy of 99.2%.

Once you draw an image it is scaled to the appropriate 28 x 28 pixel resolution and fed into both neural networks.

As you will discover, the convolutional neural network will prove more accurate at this image classification task.

Shafeen Tejani

Figure 7: Both networks have a difficult time estimating values drawn on the edges of the area

NOTE

In this example, we see the number **1** drawn to the right side of the drawing area. The probability of this number being a **1** is **0** in both networks.

The MNIST dataset does not contain numbers on the edges of images. Hence, neither network assigns relevant values to the pixels located in that region. Both networks are much better at classifying numbers correctly if we draw them closer to the center of the designated area. This shows that neural networks can only be as powerful as the data that is used to train them. If the data used for training is very different than what we are trying to predict, the network will most likely produce disappointing results.

Activity 2 – Exploring a Trained Neural Network

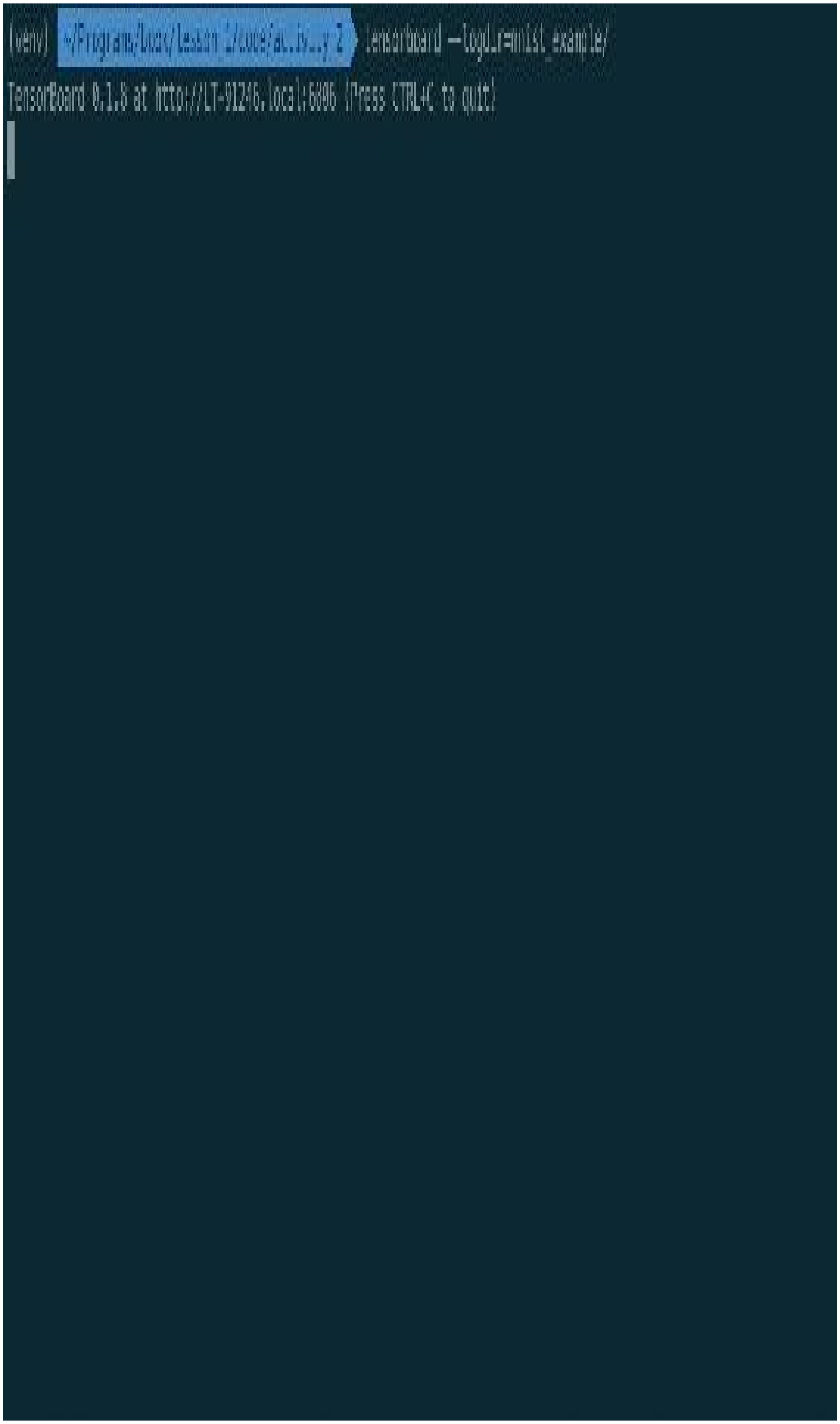
In this section, we will explore the neural network that we have trained during our exercis. We will also train a few other networks by altering hyperparameters from our original one.

Let's start by exploring the network trained in our exercise. We have provided that same trained network as binary files in the directory of this book. Let's open that trained network using TensorBoard and explore its components.

Using your terminal, navigate to the directory `lesson_1/activity_2` and execute the following command to start TensorBoard:

```
$ tensorboard --logdir=mnist_example/
```

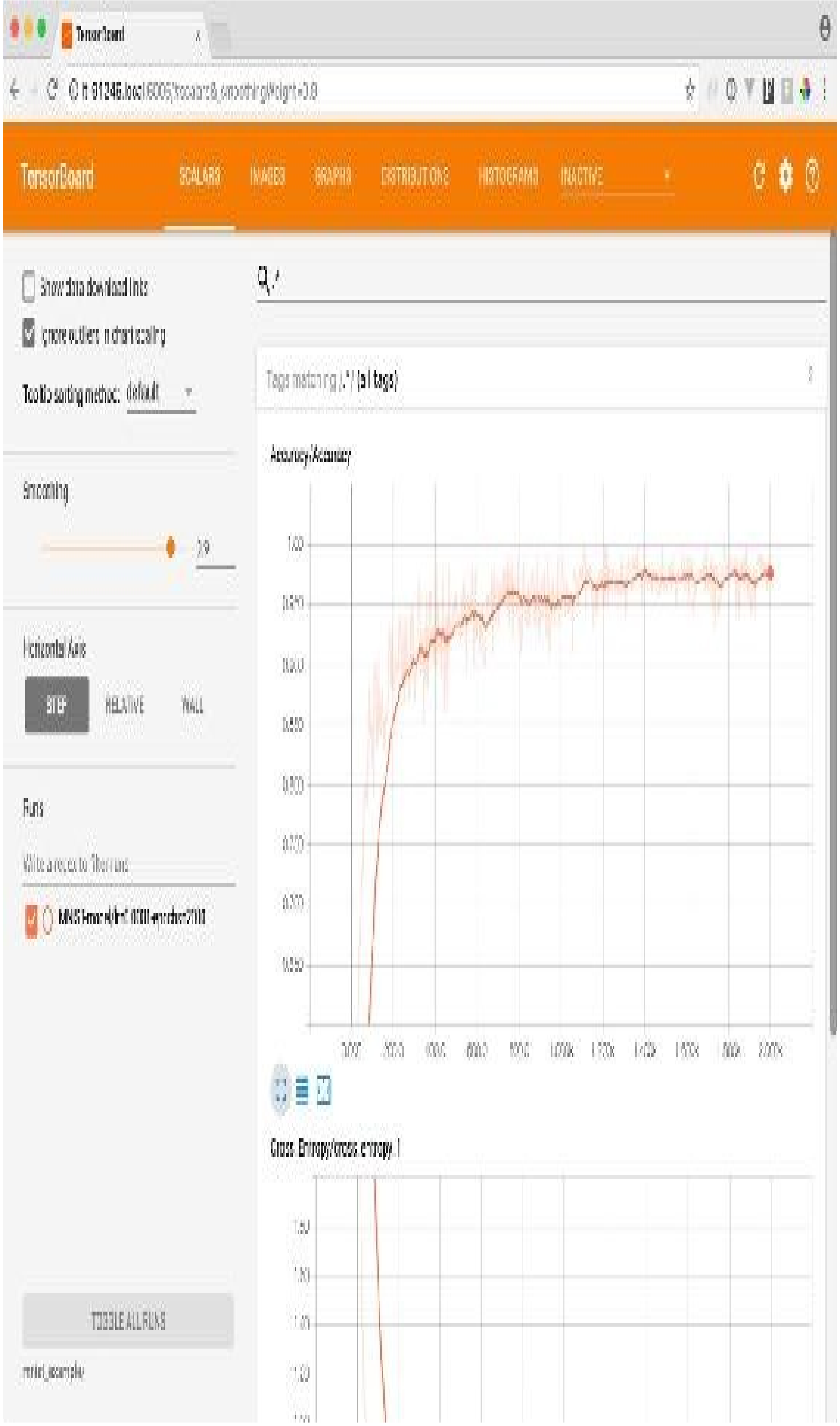
Now, open the URL provided by TensorBoard in your browser. You should be able to see the TensorBoard scalars page:



```
(venv) ~/Programs/book/Lesson_2/code/activity_2 tensorboard --logdir=mnist_example/
TensorBoard 0.1.8 at http://LT-91246.local:6006 (Press CTRL+C to quit)
```

Figure 8: Image of a terminal after starting a TensorBoard instance

After you open the URL provided by the `tensorboard` command, you should be able to see the following TensorBoard page:



□ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.9

Horizontal Axis: [✓ STEP] RELATIVE WALL

Runs

Write a regex to filter runs

☑ ○ MNIST_model/lr=1.000 epochs=2000

mnist_example

Tags matching /.*/ (all tags)

Figure 9: Image of the TensorBoard landing page

Let's now explore our trained neural network and see how it performed.

On the TensorBoard page, click on the **Scalars** page and enlarge the **Accuracy** graph. Now, move the **Smoothing** slider to **0.9**.

The accuracy graph measures how accurate the network was able to guess the labels of a test set. At first, the network guesses those labels completely wrong. This happens because we have initialized the weights and biases of our network with random values, so its first attempts are a guess. The network will then change the weights and biases of its layers on a second run; the network will continue to invest in the nodes that give positive results by altering their weights and biases, and penalize those that don't by gradually reducing their impact on the network (eventually reaching `0`). As you can see, this is a really efficient technique that quickly yields great results.

Let's focus our attention on the **Accuracy** graph. See how the algorithm manages to reach great accuracy (> 95 percent) after around 1,000 epochs? What happens between 1,000 and 2,000 epochs? Would it get more accurate if we continued to train with more epochs?

Between 1,000 and 2,000 is when the accuracy of the network continues to improve, but at a decreasing rate. The network may improve slightly if trained with more epochs, but it will not reach 100 percent accuracy with the current architecture.

The script is a modified version of an official Google script that was created to show how TensorFlow works. We have divided the script into functions that are easier to understand and added many comments to guide your learning. Try running that script by modifying the variables at the top of the script:

```
LEARNING_RATE = 0.0001
 EPOCHS = 2000
```

NOTE

Use the `mnist.py` file for your reference at `Code/Lesson-1/activity_2/`.

Now, try running that script by modifying the values of those variables. For instance, try modifying the learning rate to `0.1` and the epochs to `100`. Do you think the network can achieve comparable results?

NOTE

There are many other parameters that you can modify in your neural network. For now, experiment with the epochs and the learning rate of your network. You will notice that those two on their own can greatly change the output of your network—but only by so much. Experiment to see if you can train this network faster with the current architecture just by altering those two parameters.

Verify how your network is training using TensorBoard. Alter those parameters a few more times by multiplying the starting values by 10 until you notice that the network is improving. This process of tuning the network and finding improved accuracy is similar to what is used in industry applications today to improve existing neural network models.

Summary

In this lesson, we explored a TensorFlow-trained neural network using TensorBoard and trained our own modified version of that network with different epochs and learning rates. This gave you hands-on experiences on how to train a highly performant neural network and also allowed you to explore some of its limitations.

Do you think we can achieve similar accuracy with real Bitcoin data? We will attempt to predict future Bitcoin prices using a common neural network algorithm during *Lesson 2*, *Model Architecture*. In *Lesson 3*, *Model Evaluation and Optimization,* we will evaluate and improve that model and, finally, in *Lesson 4*, *Productization,* we will create a program that serves the prediction of that system via a HTTP API.

Chapter 2. Model Architecture

Building on fundamental concepts from *Lesson 1*, *Introduction to Neural Networks and Deep Learning,* we now move into a practical problem: can we predict Bitcoin prices using a deep learning model? In this lesson, we will learn how to build a deep learning model that attempts to do that.

We will conclude this lesson by putting all of these components together and building a bare-bones yet complete first version of a deep learning application.

Lesson Objectives

In this lesson, you will:

- Prepare data for a deep learning model
- Choose the right model architecture
- Use Keras, a TensorFlow abstraction library
- Make predictions with a trained model

Choosing the Right Model Architecture

Deep learning is a field undergoing intense research activity. Among other things, researchers are devoted to inventing new neural network architectures that can either tackle new problems or increase the performance of previously implemented architectures.

In this section, we study both old and new architectures. Older architectures have been used to solve a large array of problems and are generally considered the right choice when starting a new project. Newer architectures have shown great successes in specific problems, but are harder to generalize. The latter are interesting as references of what to explore next, but are hardly a good choice when starting a project.

Common Architectures

Considering the many architecture possibilities, there are two popular architectures that have often been used as starting points for a number of applications: **convolutional neural networks** (**CNNs**) and **recurrent neural networks** (**RNNs**). These are foundational networks and should be considered starting points for most projects. We also include descriptions of another three networks, due to their relevance in the field: **Long-**

short term memory (**LSTM**) networks, an RNN variant; **generative adversarial networks** (**GANs**); and deep reinforcement learning. These latter architectures have shown great successes in solving contemporary problems, but are somewhat more difficult to use.

CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural networks have gained notoriety for working with problems that have a grid-like structure. They were originally created to classify images, but have been used in a number of other areas, ranging from speech recognition to self-driving vehicles.

CNN's essential insight is to use closely related data as an element of the training process, instead of only individual data inputs. This idea is particularly effective in the context of images, where a pixel located to the right of another pixel is related to that pixel as well, given that they form part of a larger composition. In this case, that composition is what the network is training to predict. Hence, combining a few pixels together is better than using an individual pixel on its own.

The name **convolution** is given to the mathematical representation of this process:

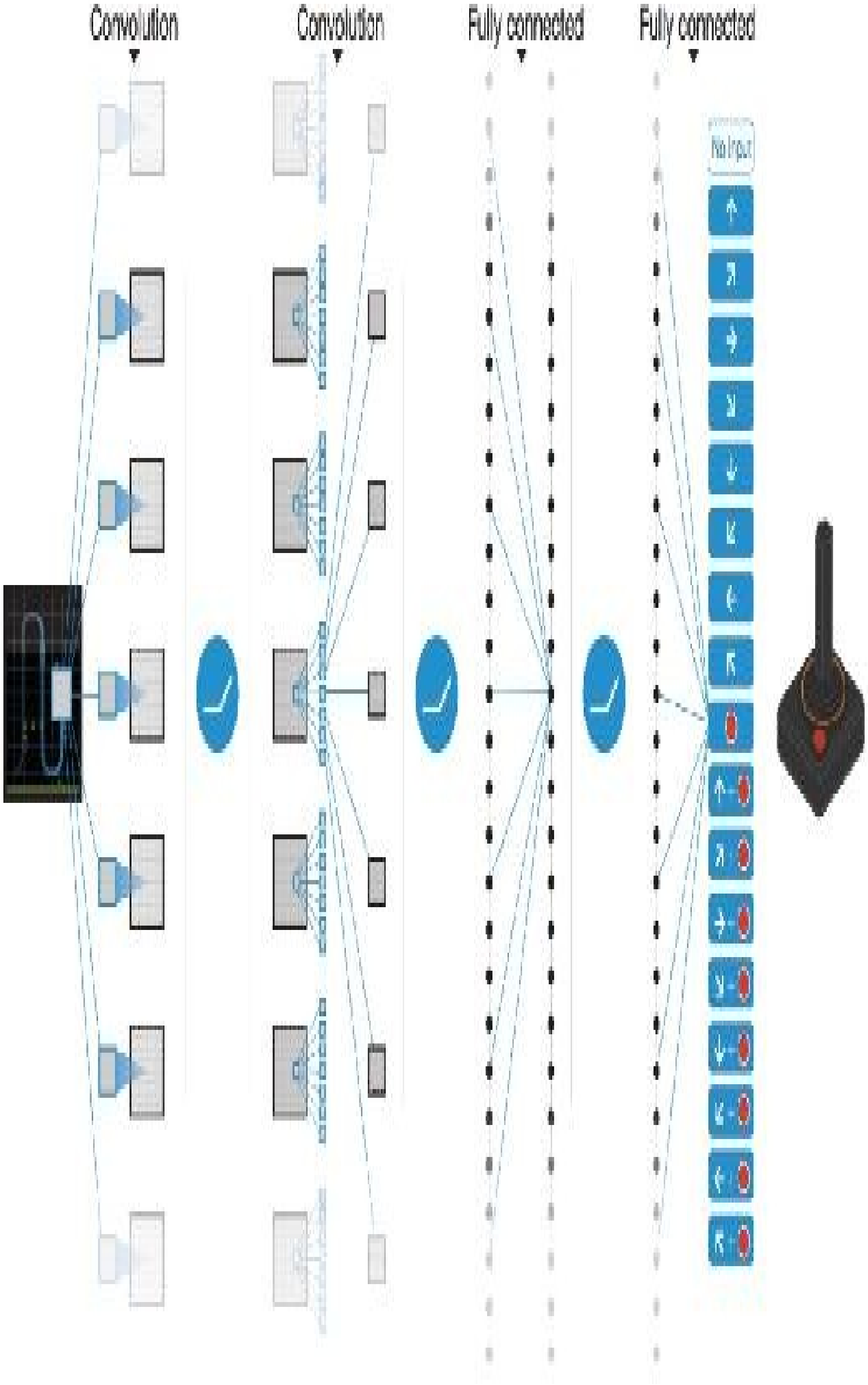


Figure 1: Illustration of the convolution process Image source: Volodymyr Mnih, et al.

NOTE

For more information refer, *Human-level control through deep reinforcement learning*. February 2015, Nature. Available at: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf.

RECURRENT NEURAL NETWORKS

Convolutional neural networks work with a set of inputs that keep altering the weights and biases of the networks' respective layers and nodes. A known limitation of this approach is that its architecture ignores the sequence of these inputs when determining how to change the networks' weights and biases.

Recurrent neural networks were created precisely to address that problem. RNNs are designed to work with sequential data. This means that at every epoch, layers can be influenced by the output of previous layers. The memory of previous observations in a given sequence plays a role in the evaluation of posterior observations.

RNNs have had successful applications in speech recognition due to the sequential nature of that problem. Also, they are used for translation problems. Google Translate's current algorithm—called **Transformer**—uses an RNN to translate text from one language to another.

NOTE

For more information refer, *Transformer: A Novel Neural Network Architecture for Language Understanding*, by Jakob Uszkoreit, Google Research Blog, August 2017. Available at: https://research.googleblog.com/2017/08/transformer-novel-neural-network.html.

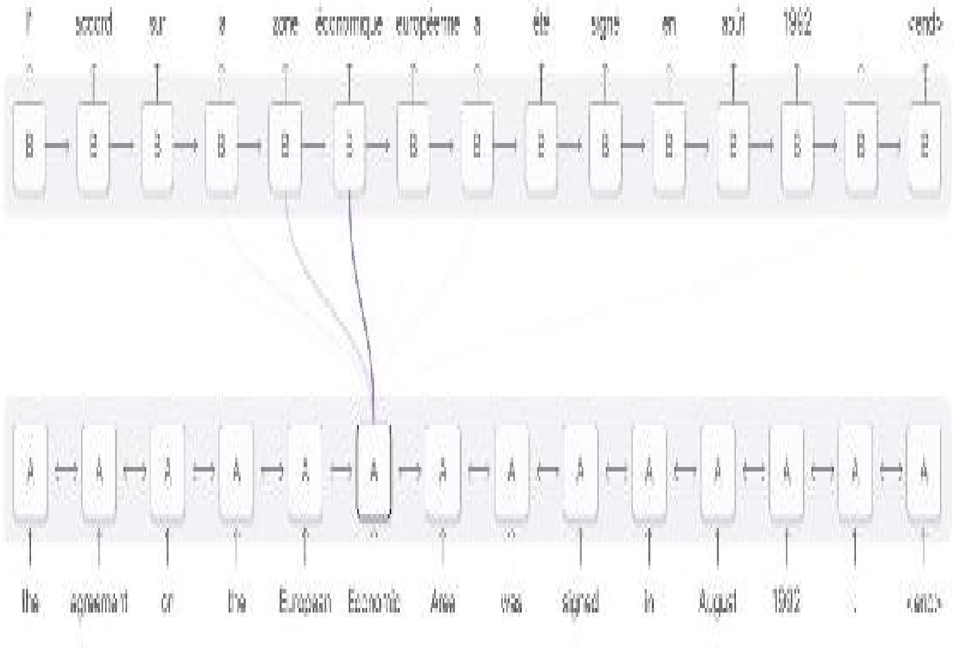


Figure 2: Illustration from distill.pub (https://distill.pub/2016/augmented-rnns/)

Figure 2 shows that words in English are related to words in French, based on where they appear in a sentence. RNNs are very popular in language translation problems.

Long-short term memory networks are RNN variants created to address the vanishing gradient problem. The vanishing gradient problem is caused by memory components that are too distant from the current step and would receive lower weights due to their distance. LSTMs are a variant of RNNs that contain a memory component—called **forget gate**. That component can be used to evaluate how both recent and old elements affect the weights and biases, depending on where the observation is placed in a sequence.

NOTE

For more details refer, The LSTM architecture was first introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997. Current implementations have had several modifications. For a detailed mathematical explanation of how each component of an LSTM works, we suggest the article *Understanding LSTM Networks* by Christopher Olah, August 2015, available at http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

GENERATIVE ADVERSARIAL NETWORKS

Generative adversarial networks (**GANs**) were invented in 2014 by Ian Goodfellow and his colleagues at the University of Montreal. GANs suggest that, instead of having one neural network that optimizes weights and biases with the objective to minimize its errors, there should be two neural networks that compete against each other for that purpose.

NOTE

For more details refer, *Generative Adversarial Networks* by Ian Goodfellow, et al, arXiv. June 10, 2014. Available at: https://arxiv.org/abs/1406.2661.

GANs have a network that generates new data (that is, "fake" data) and a network that evaluates the likelihood of the data generated by the first network to be real or "fake". They compete because both learn: one learns how to better generate "fake" data, and the other learns how to distinguish if the data it is presented with is real or not. They iterate on every epoch until they both converge. That is the point when the network that evaluates generated data cannot distinguish between "fake" and real data any longer.

GANs have been successfully used in fields where data has a clear topological structure. Its original implementation used a GAN to create synthetic images of objects, people's faces, and animals that were similar to real images of those things. This domain of image creation is where GANs are used the most frequently, but applications in other domains occasionally appear in research papers.

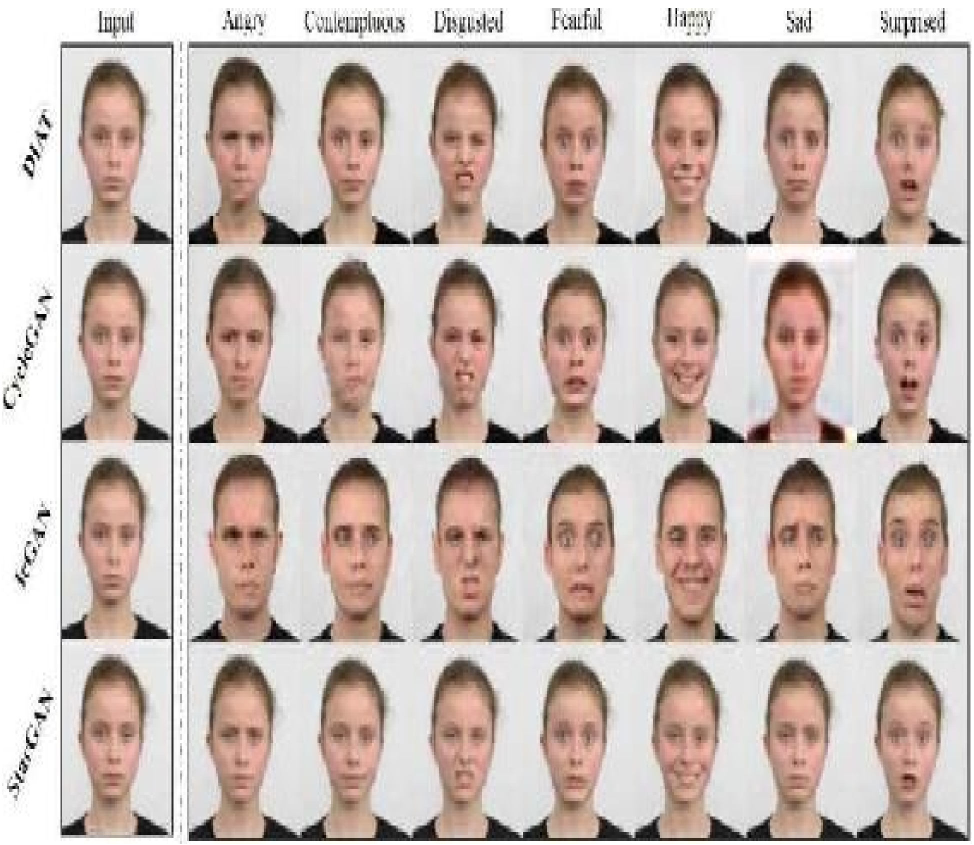


Figure 3: Image that shows the result of different GAN algorithms in changing people's faces based on a given emotion. Source: StarGAN Project. Available at https://github.com/yunjey/StarGAN.

DEEP REINFORCEMENT LEARNING

The original DRL architecture was championed by DeepMind, a Google-owned artificial intelligence research organization based in the UK. The key idea of DRL networks is that they are unsupervised in nature and that they learn from trial-and-error, only optimizing for a reward function. That is, different than other networks (which use supervised approaches to optimize for how wrong the predictions are, compared to what is known to be right), DRL networks do not know of a correct way of approaching a problem. They are simply given the rules of a system and are then rewarded every time they perform a function correctly. This process, which takes a very large number of iterations, eventually trains networks to excel in a number of tasks.

NOTE

For more information refer, *Human-level control through deep reinforcement learning*, by Volodymyr Mnih et al., February 2015, Nature. Available at: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf.

Deep Reinforcement Learning models gained popularity after DeepMind created AlphaGo, a system that plays the game Go better than professional players. DeepMind also created DRL networks that learn how to play video games at a superhuman level, entirely on their own:

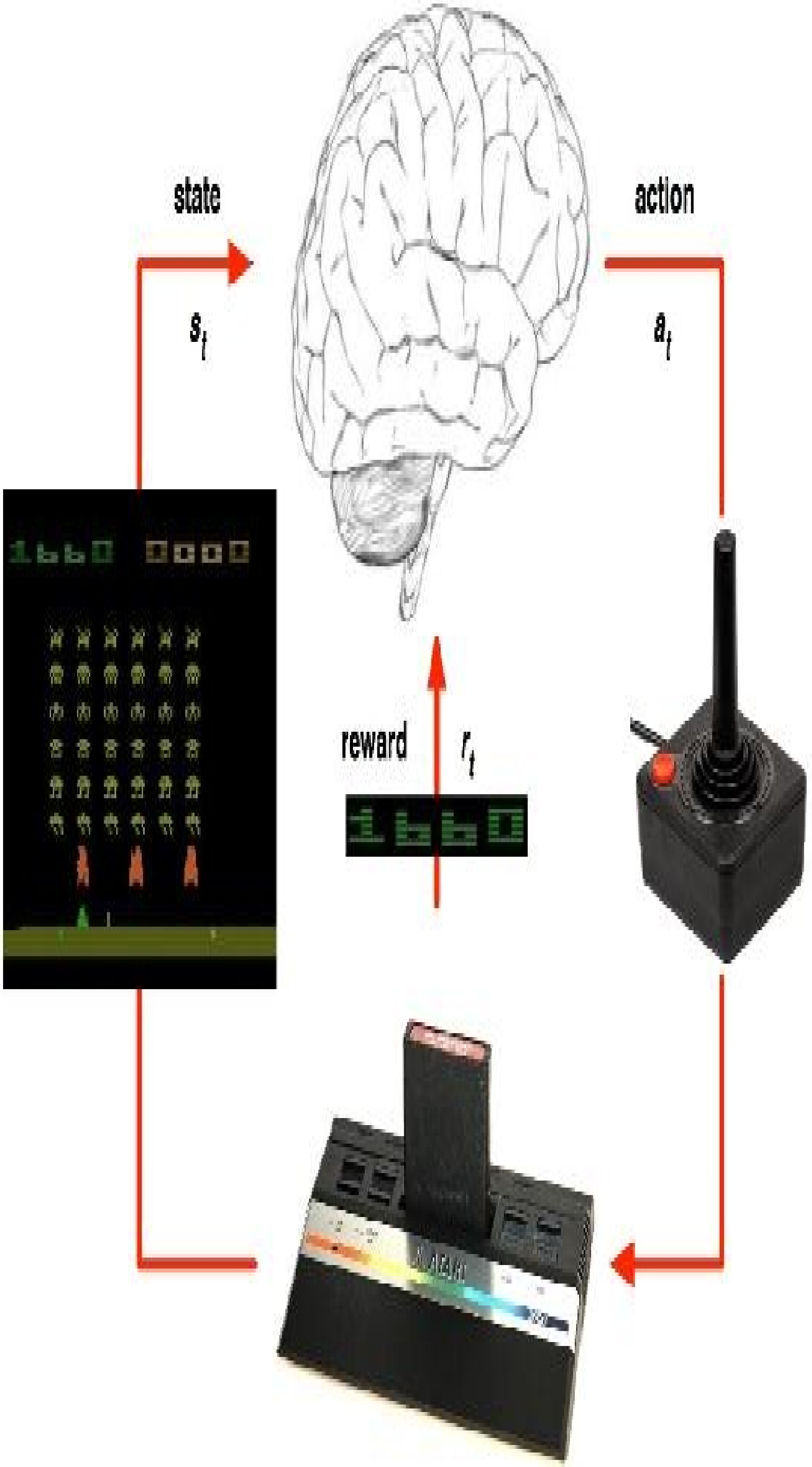

state
s_t
action
a_t
reward
r_t
1660
ATARI

Figure 4: Image that represents how the DQN algorithm works

NOTE

For more information refer, DQN was created by DeepMind to beat Atari games. The algorithm uses a deep reinforcement learning solution to continuously increase its reward. Image source: https://keon.io/deep-q-learning/.

Architecture	**Data Structure**	**Successful Applications**
Convolutional neural networks (CNNs)	Grid-like topological structure (that is, images)	Image recognition and classification
Recurrent neural network (RNN) and long-short term memory (LSTM) networks	Sequential data (that is, time-series data)	Speech recognition, text generation, and translation
Generative adversarial networks (GANs)	Grid-like topological structure (that is, images)	Image generation
Deep reinforcement learning (DRL)	System with clear rules and a clearly defined reward function	Playing video games and self-driving vehicles

Table 1: Different neural network architectures have shown success in different fields. The networks' architecture is typically related to the structure of the problem at hand.

Data Normalization

Before building a deep learning model, one more step is necessary: data normalization.

Data normalization is a common practice in machine learning systems. Particularly regarding neural networks, researchers have proposed that normalization is an essential technique for training RNNs (and LSTMs), mainly because it decreases the network's training time and increases the network's overall performance.

NOTE

For more information refer, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* by Sergey Ioffe et. al., arXiv, March 2015. Available at: https://arxiv.org/abs/1502.03167.

Deciding on a normalization technique varies, depending on the data and the problem at hand. The following techniques are commonly used.

Z-SCORE

When data is normally distributed (that is, Gaussian), one can compute the distance between each observation as a standard deviation from its mean. This normalization is useful when identifying how distant data points are from more likely occurrences in the distribution. The Z-score is defined by:

$$z_i = \frac{x_i - \mu}{\sigma}$$

Here, x_i is the i^{th} observation, μ the mean, and σ the standard deviation of the series.

NOTE

For more information refer, Standard score article (Z-score). Wikipedia. Available at: https://en.wikipedia.org/wiki/Standard_score.

POINT-RELATIVE NORMALIZATION

This normalization computes the difference of a given observation in relation to the first observation of the series. This kind of normalization is useful to identify trends in relation to a starting point. The point-relative normalization is defined by:

$$n_i = \left(\frac{o_i}{o_0}\right) - 1$$

Here, o_i is the i^{th} observation and o_0 is the first observation of the series.

NOTE

As suggested by Siraj Raval in his video, *How to Predict Stock Prices Easily - Intro to Deep Learning #7,* available on YouTube at: https://www.youtube.com/watch?v=ftMq5ps503w.

MAXIMUM AND MINIMUM NORMALIZATION

This normalization computes the distance between a given observation and the maximum and minimum values of the series. This normalization is useful when working with series in which the maximum and minimum values are not outliers and are important for future predictions. This normalization technique can be applied with:

$$n_i = \frac{\mathrm{o_i} - \min(O)}{\max(O) - \min(O)}$$

Here, o_i is the i^{th} observation, *O* represents a vector with all O values, and the functions min (O) and max (O) represent the minimum and maximum values of the series, respectively.

During *Activity 3*, *Exploring the Bitcoin Dataset and Preparing Data for Model,* we will prepare available Bitcoin data to be used in our LSTM mode. That includes selecting variables of interest, selecting a relevant period, and applying the preceding point-relative normalization technique.

Structuring Your Problem

Compared to researchers, practitioners spend much less time determining which architecture to choose when starting a new deep learning project. Acquiring data that represents a given problem correctly is the most important factor to consider when developing these systems, followed by the understanding of the dataset's inherent biases and limitations.

When starting to develop a deep learning system, consider the following questions for reflection:

- **Do I have the right data?** This is the hardest challenge when training a deep learning model. First, define your problem with mathematical rules. Use precise definitions and organize the problem in either categories (classification problems) or a continuous scale (regression problems). Now, how can you collect data about those metrics?
- **Do I have enough data?** Typically, deep learning algorithms have shown to perform much better in large datasets than in smaller ones. Knowing how much data is necessary to train a high-performance algorithm depends on the kind of problem you are trying to address, but aim to collect as much data as you can.
- **Can I use a pre-trained model?** If you are working on a problem that is a subset of a more general application—but within the same domain—consider using a pre-trained model. Pre-trained models can give you a head start on tackling the specific patterns of your problem,

instead of the more general characteristics of the domain at large. A good place to start is the official TensorFlow repository (https://github.com/tensorflow/models).

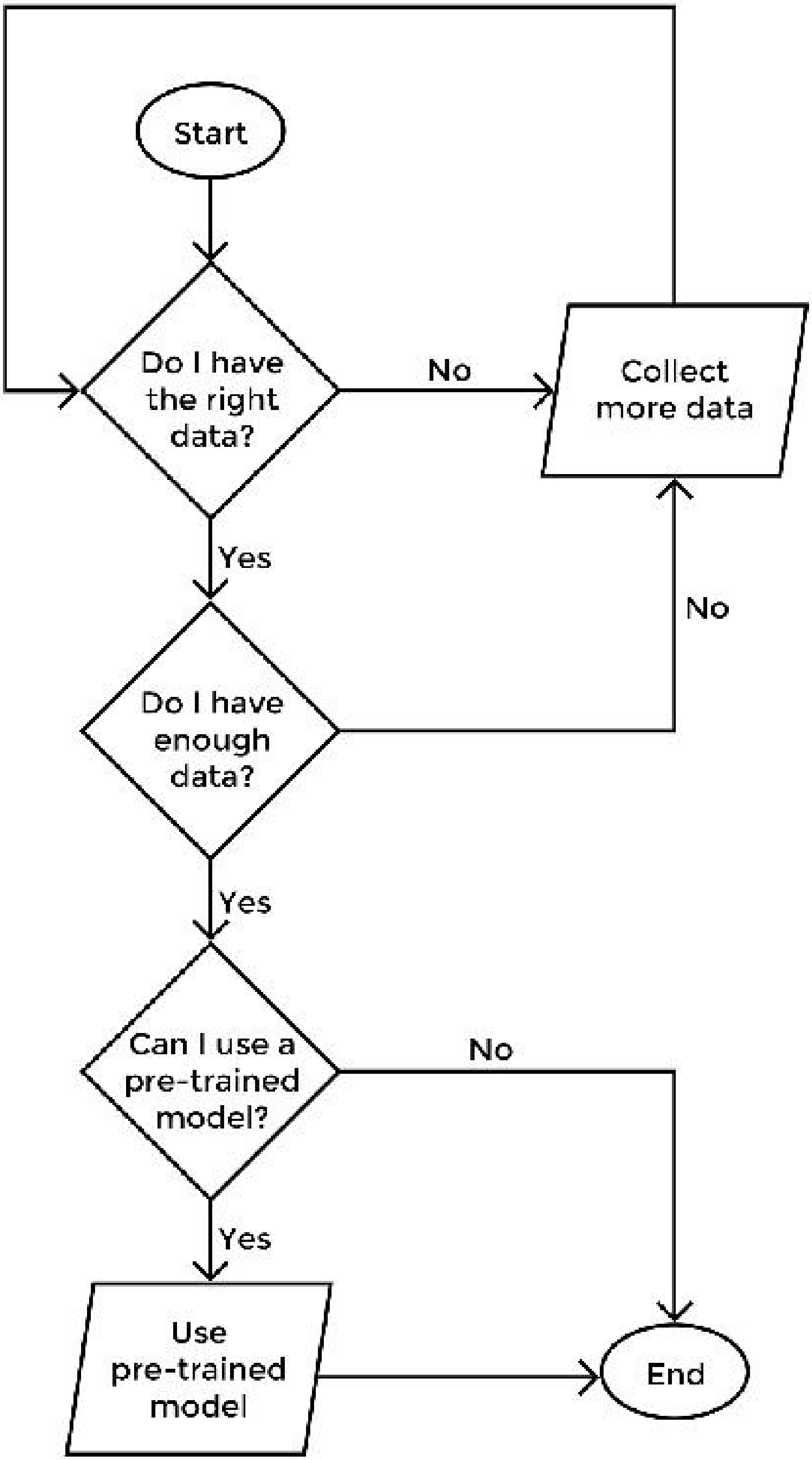

Start
Do I have the right data?
No
Collect more data
Yes
No
Do I have enough data?
Yes
Can I use a pre-trained model?
No
Yes
Use pre-trained model
End

Figure 5: Decision-tree of key reflection questions to be made at the beginning of a deep learning project

In certain circumstances, data may simply not be available. Depending on the case, it may be possible to use a series of techniques to effectively create more data from your input data. This process is known as **data augmentation** and has successful application when working with image recognition problems.

> **NOTE**
>
> A good reference is the article *Classifying plankton with deep neural networks* available at http://benanne.github.io/2015/03/17/plankton.html. The authors show a series of techniques for augmenting a small set of image data in order to increase the number of training samples the model has.

Activity 3 – Exploring the Bitcoin Dataset and Preparing Data for Model

We will be using a public dataset originally retrieved from CoinMarketCap, a popular website that tracks different cryptocurrency statistics. The dataset has been provided alongside this lesson and will be used throughout the rest of this book.

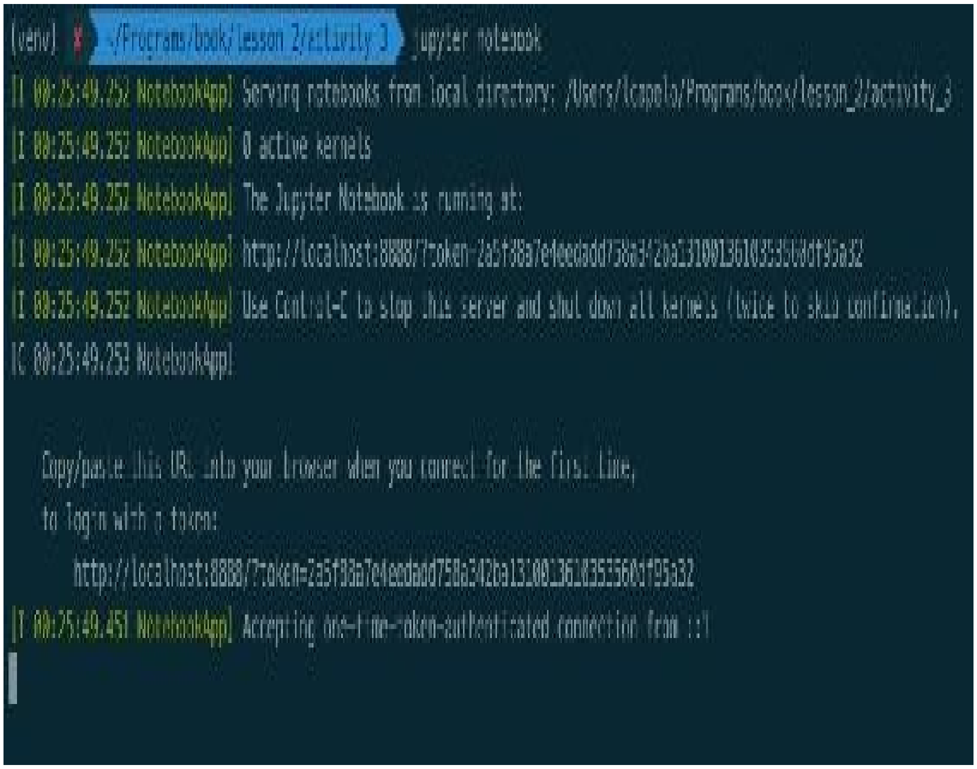
We will be exploring the dataset using Jupyter Notebooks. Jupyter Notebooks provide Python sessions via a web-browser that allows you to work with data interactively. They are a popular tool for exploring datasets. They will be used in activities throughout this book.

Using your terminal, navigate to the directory `lesson_2/activity_3` and execute the following command to start a Jupyter Notebook instance:

```
$ jupyter notebook
```

Now, open the URL provided by the application in your browser. You should be able to see a Jupyter Notebook page with a number of directories from your file system.

You should see the following output:



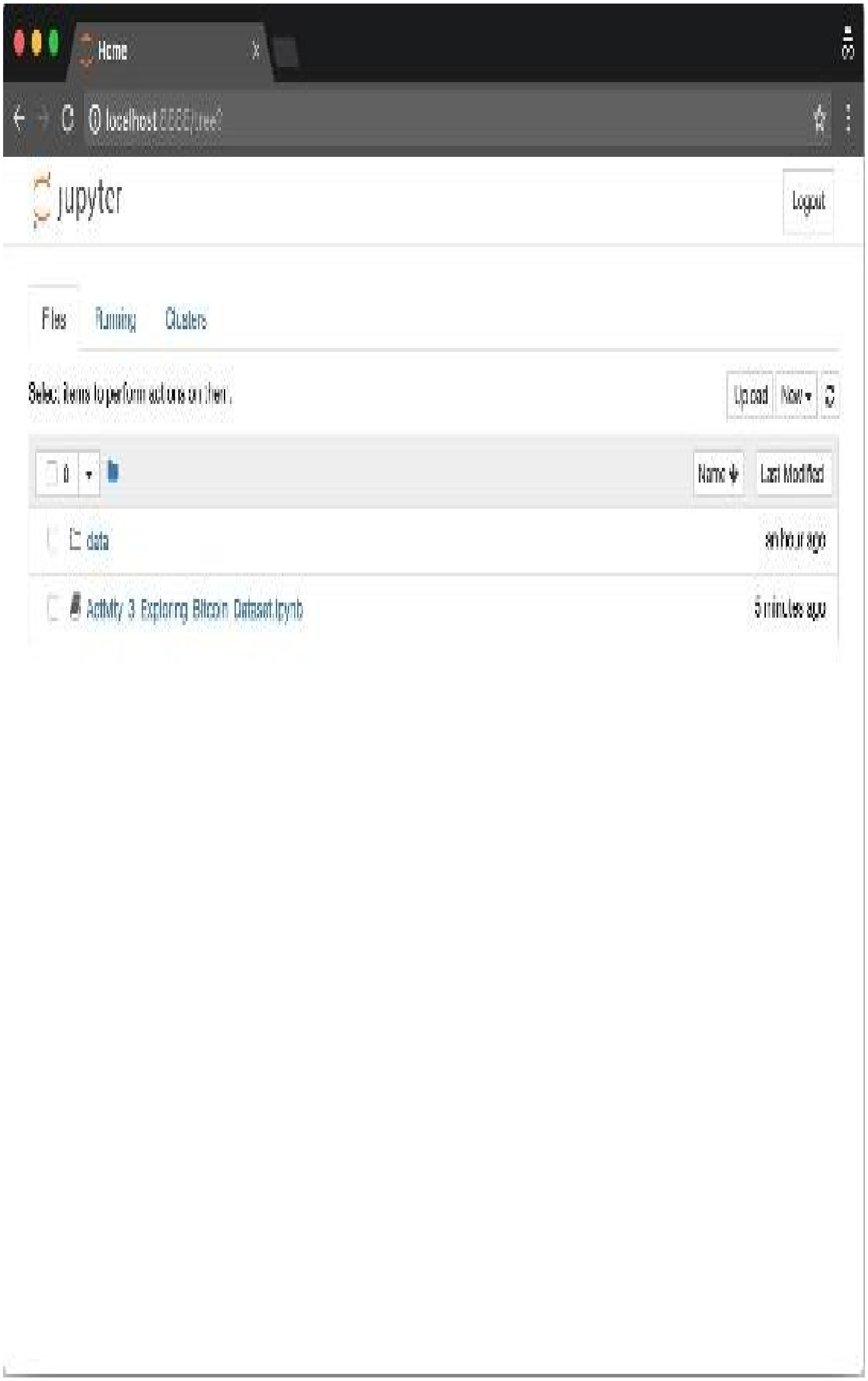
```
(venv) ✘ ~/Programs/book/lesson_2/activity_3  jupyter notebook
[I 00:25:49.252 NotebookApp] Serving notebooks from local directory: /Users/lcapelo/Programs/book/lesson_2/activity_3
[I 00:25:49.252 NotebookApp] 0 active kernels
[I 00:25:49.252 NotebookApp] The Jupyter Notebook is running at:
[I 00:25:49.252 NotebookApp] http://localhost:8888/?token=2a5f8a7e4eedadd758a342ba13100136103535560df95a32
[I 00:25:49.252 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 00:25:49.253 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=2a5f8a7e4eedadd758a342ba13100136103535560df95a32
[I 00:25:49.451 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```

Figure 6: Terminal image after starting a Jupyter Notebook instance. Navigate to

the URL show in a browser, and you should be able to see the Jupyter Notebook landing page.

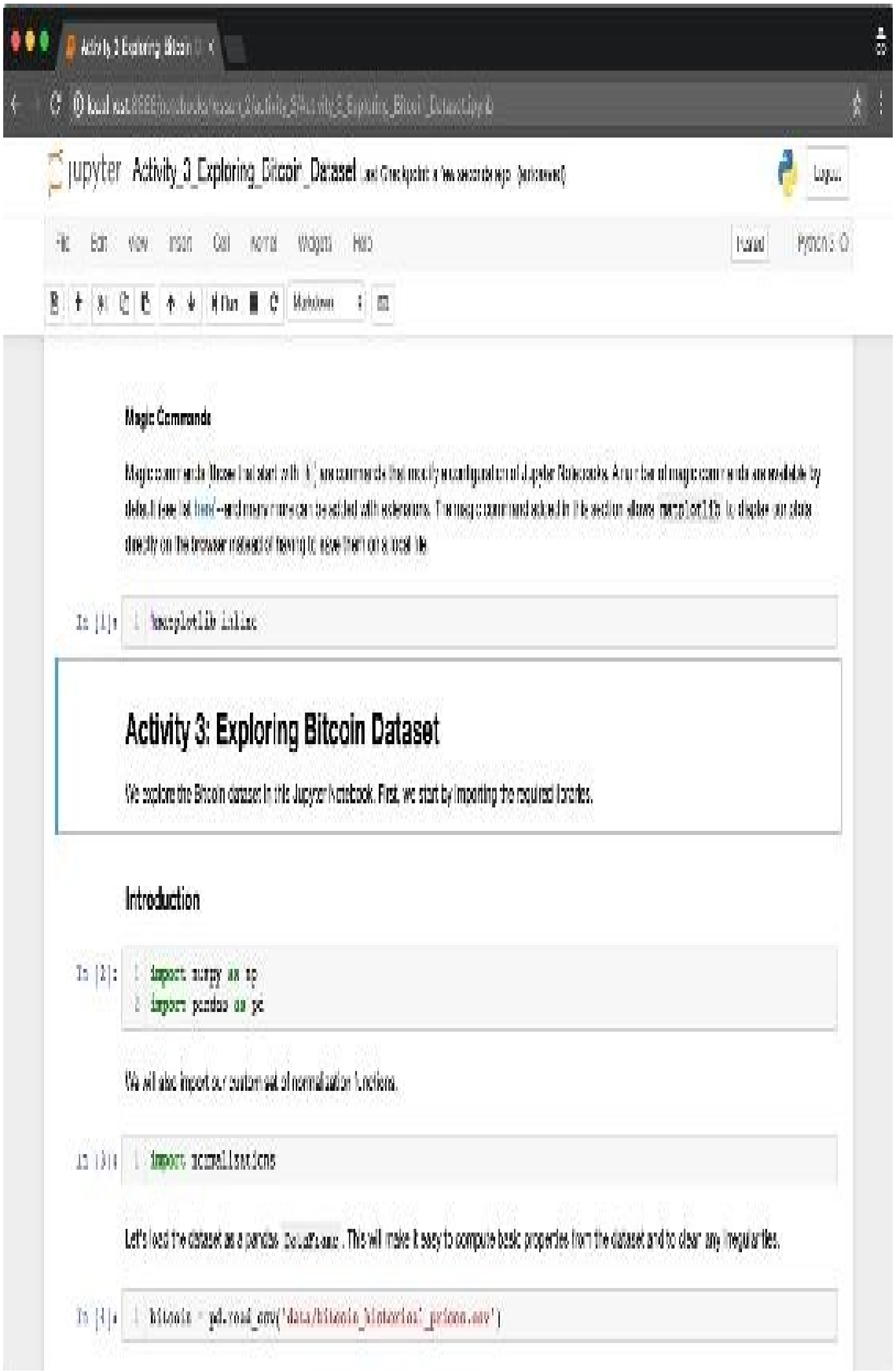
Now, navigate to the directories and click on the file `Activity_3_Exploring_Bitcoin_Dataset.ipynb` . This is a Jupyter Notebook file that will be opened in a new browser tab. The application will automatically start a new Python interactive session for you.



Select items to perform actions on them.

□ data	an hour ago
□ Activity 3 Exploring Bitcoin Dataset.ipynb	5 minutes ago

Figure 7: Landing page of your Jupyter Notebook instance


Activity 3 Exploring Bitcoin D
localhost:8888/notebooks/lesson_2/activity_3/Activity_3_Exploring_Bitcoin_Dataset.ipynb
jupyter
Activity_3_Exploring_Bitcoin_Dataset Last Checkpoint: a few seconds ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Markdown

Magic Commands

Magic commands (those that start with `%`) are commands that modify a configuration of Jupyter Notebooks. A number of magic commands are available by default (see list here)—and many more can be added with extensions. The magic command added in this section allows `matplotlib` to display our plots directly on the browser instead of having to save them on a local file.

```
In [1]: %matplotlib inline
```

Activity 3: Exploring Bitcoin Dataset

We explore the Bitcoin dataset in this Jupyter Notebook. First, we start by importing the required libraries.

Introduction

```
In [2]: import numpy as np
        import pandas as pd
```

We will also import our custom set of normalization functions.

```
In [3]: import normalizations
```

Let's load the dataset as a pandas `DataFrame`. This will make it easy to compute basic properties from the dataset and to clean any irregularities.

```
In [4]: bitcoin = pd.read_csv('data/bitcoin_historical_prices.csv')
```

Figure 8: Image of the Notebook Activity_3_Exploring_Bitcoin_Dataset.ipyn

b. You can now interact with that

Notebook and make modifications.

After opening our Jupyter Notebook, let's now explore the Bitcoin data made available with this lesson.

The dataset `data/bitcoin_historical_prices.csv` contains measurements of Bitcoin prices since early 2013. The most recent observation is on November 2017—the dataset comes from CoinMarketCap, an online service that is updated daily. It contains eight variables, two of which (`date` and `week`) describe a time period of the data—these can be used as indices—and six others (`open`, `high`, `low`, `close`, `volume`, and `market_capitalization`) that can be used to understand how the price and value of Bitcoin has changed over time:

Variable	**Description**
`date`	Date of the observation.
`iso_week`	Week number for a given year.
`open`	Open value for a single Bitcoin coin.
`high`	Highest value achieved during a given day period.

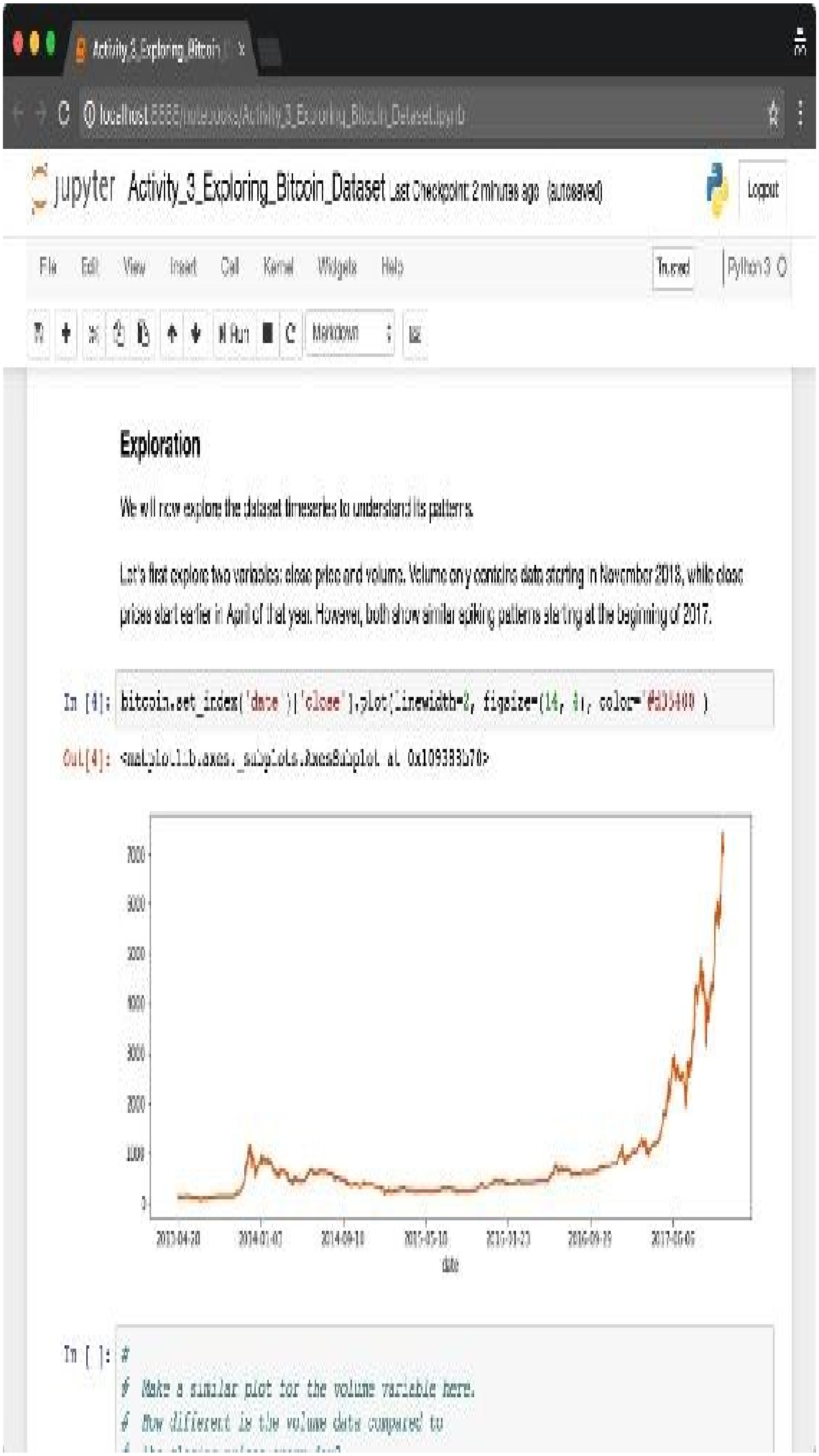
`low`	Lowest value achieved during a given day period.
`close`	Value at the close of the transaction day.
`volume`	The total volume of Bitcoin that was exchanged during that day.
`market_capitalization`	Market capitalization, which is explained by *Market Cap = Price * Circulating Supply*.

Table 2: Available variables (that is, columns) in the Bitcoin historical prices dataset

Using the open Jupyter Notebook instance, let's now explore the time-series of two of those variables: `close` and `volume`. We will start with those time-series to explore price-fluctuation patterns.

Navigate to the open instance of the Jupyter Notebook `Activity_3_Exploring_Bitcoin_Dataset.ipynb`. Now, execute all cells under the header **Introduction**. This will import the required libraries and import the dataset into memory.

After the dataset has been imported into memory, move to the **Exploration** section. You will find a snippet of code that generates a time-series plot for the `close` variable. Can you generate the same plot for the `volume` variable?



Exploration

We will now explore the dataset timeseries to understand its patterns.

Let's first explore two variables: close price and volume. Volume only contains data starting in November 2013, while close prices start earlier in April of that year. However, both show similar spiking patterns starting at the beginning of 2017.

```
In [4]: bitcoin.set_index('date')['close'].plot(linewidth=2, figsize=(14, 4), color='#d35400')
```

```
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x109333b70>
```

```
In [ ]: #
        # Make a similar plot for the volume variable here.
        # How different is the volume data compared to
        # the closing prices every day?
```

Figure 9: Time-series plot of the closing price for Bitcoin from the close variable. Reproduce this plot, but using the volume variable in a new cell below this one.

You will have most certainly noticed that both variables surge in 2017. This reflects the current phenomenon that both the prices and value of Bitcoin have been continuously growing since the beginning of that year.

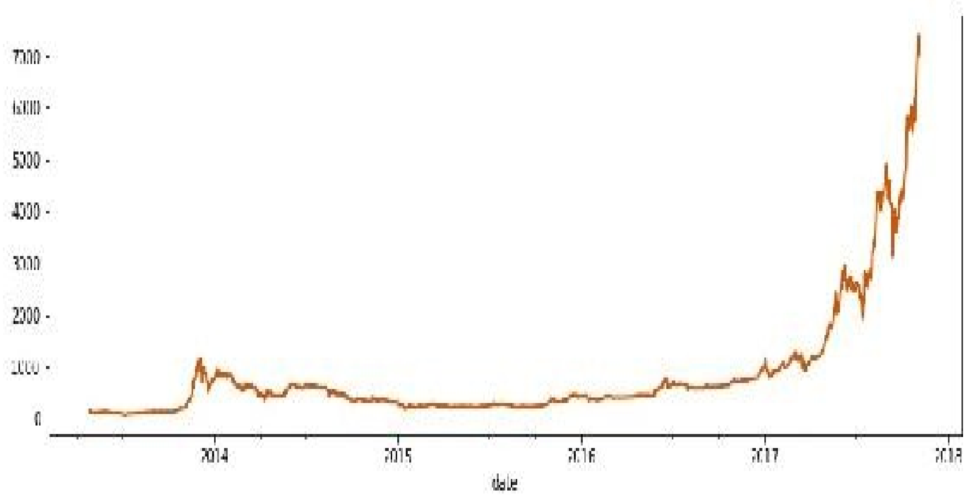


Figure 10: Closing price of Bitcoin coins in USD. Notice an early spike by late 2013 and early 2014. Also, notice how the recent prices have skyrocketed since the beginning of 2017.

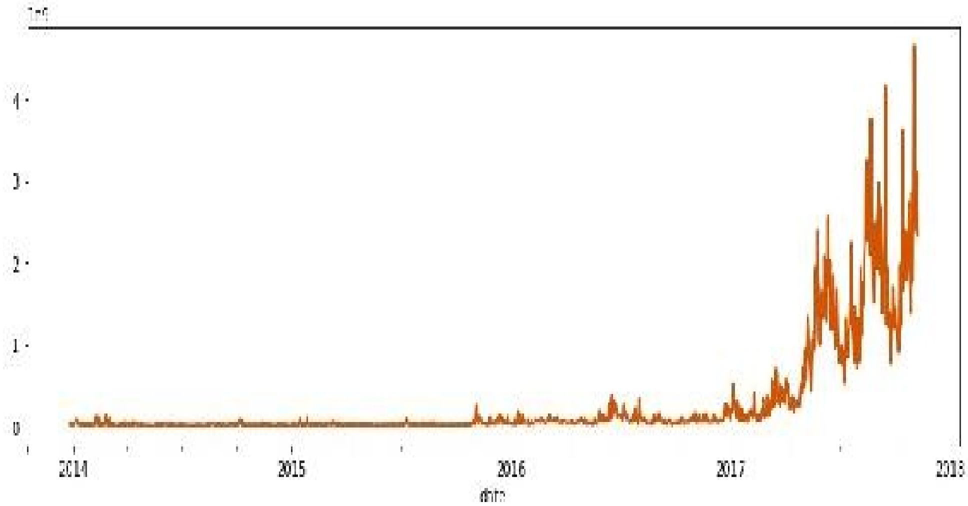


Figure 11: The volume of transactions of Bitcoin coins (in USD) shows that starting in 2017, a trend starts in which a significantly larger amount of Bitcoin is being transacted in the market. The total daily volume varies much more than daily closing prices.

Also, we notice that for many years, Bitcoin prices did not fluctuate as much as in recent years. While those periods can be used by a neural network to understand certain patterns, we will be excluding older observations, given that we are interested in predicting future prices for not-too-distant periods. Let's filter the data for 2016 and 2017 only.

Navigate to the **Preparing Dataset for Model** section. We will use the pandas API for filtering the data for the years 2016 and 2017. Pandas provides an intuitive API for performing this operation:

```
bitcoin_recent = bitcoin[bitcoin['date']
>= '2016-01-01']
```

The variable `bitcoin_recent` now has a copy of our original bitcoin dataset, but filtered to the observations that are newer or equal to January 1, 2016.

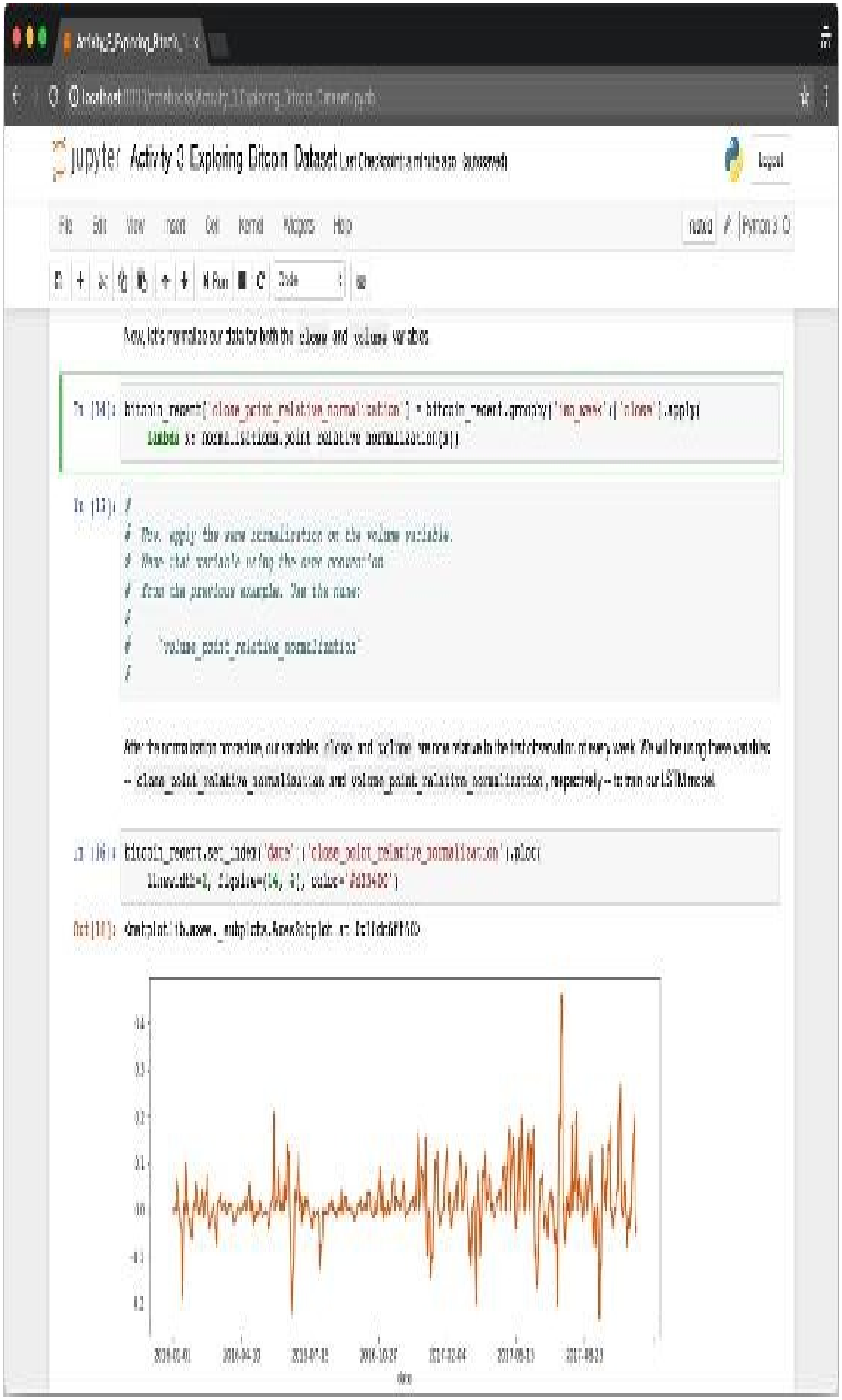
As our final step, we now normalize our data using the point-relative normalization technique described in the *Data Normalization* section. We will only normalize two variables (`close` and `volume`), because those are the variables that we are working to predict.

In the same directory containing this lesson, we have placed a script called `normalizations.py`. That script contains the three normalization techniques described in this lesson. We import that script into our Jupyter Notebook and apply the functions to our series.

Navigate to the **Preparing Dataset for Model** section. Now, use the `iso_week` variable to group all the day observations from a given week using the pandas method `groupby()`. We can now apply the normalization function `normalizations.point_relative_normalization()` directly to the series within that week. We store the output of that normalization as a new variable in the same pandas dataframe using:

```
bitcoin_recent['close_point_relative_norma
lization'] =
bitcoin_recent.groupby('iso_week')
['close'].apply(
['close'].apply(
lambda x:
normalizations.point_relative_normalizatio
n(x))
```

The variable `close_point_relative_normalization` now contains the normalized data for the variable `close`. Do the same with the variable `volume`:


Activity_3_Exploring_Bitcoin_
localhost:8888/notebooks/Activity_3_Exploring_Bitcoin_Dataset.ipynb
jupyter Activity 3 Exploring Bitcoin Dataset Last Checkpoint: a minute ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code
0.4
0.3
0.2
0.1
0.0
-0.1
-0.2
2016-02-01
2016-04-30
2016-07-19
2016-10-27
2017-02-04
2017-05-13
2017-08-21
date

Now, let's normalize our data for both the `close` and `volume` variables.

```
In [14]: bitcoin_recent['close_point_relative_normalization'] = bitcoin_recent.groupby('iso_week')['close'].apply(
             lambda x: normalizations.point_relative_normalization(x))
```

```
In [15]: #
         # Now, apply the same normalization on the volume variable.
         # Name that variable using the same convention
         # from the previous example. Use the name:
         #
         #    'volume_point_relative_normalization'
         #
```

After the normalization procedure, our variables `close` and `volume` are now relative to the first observation of every week. We will be using these variables -- `close_point_relative_normalization` and `volume_point_relative_normalization`, respectively -- to train our LSTM model.

```
In [16]: bitcoin_recent.set_index('date')['close_point_relative_normalization'].plot(
             linewidth=2, figsize=(14, 4), color='#d35400')
```

```
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x11dc6f7f60>
```

Figure 12: Image of Jupyter Notebook

focusing on section where the normalization function is applied

The normalized `close` variable contains an interesting variance pattern every week. We will be using that variable to train our LSTM model.

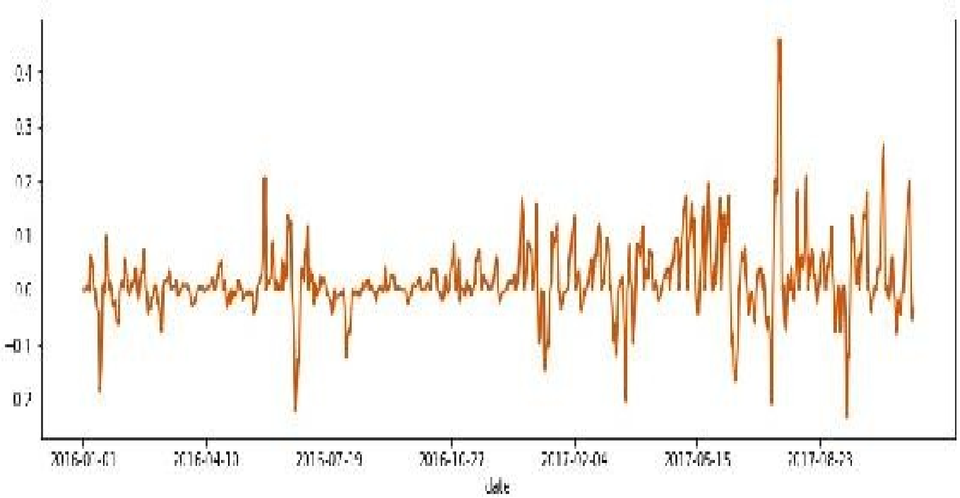
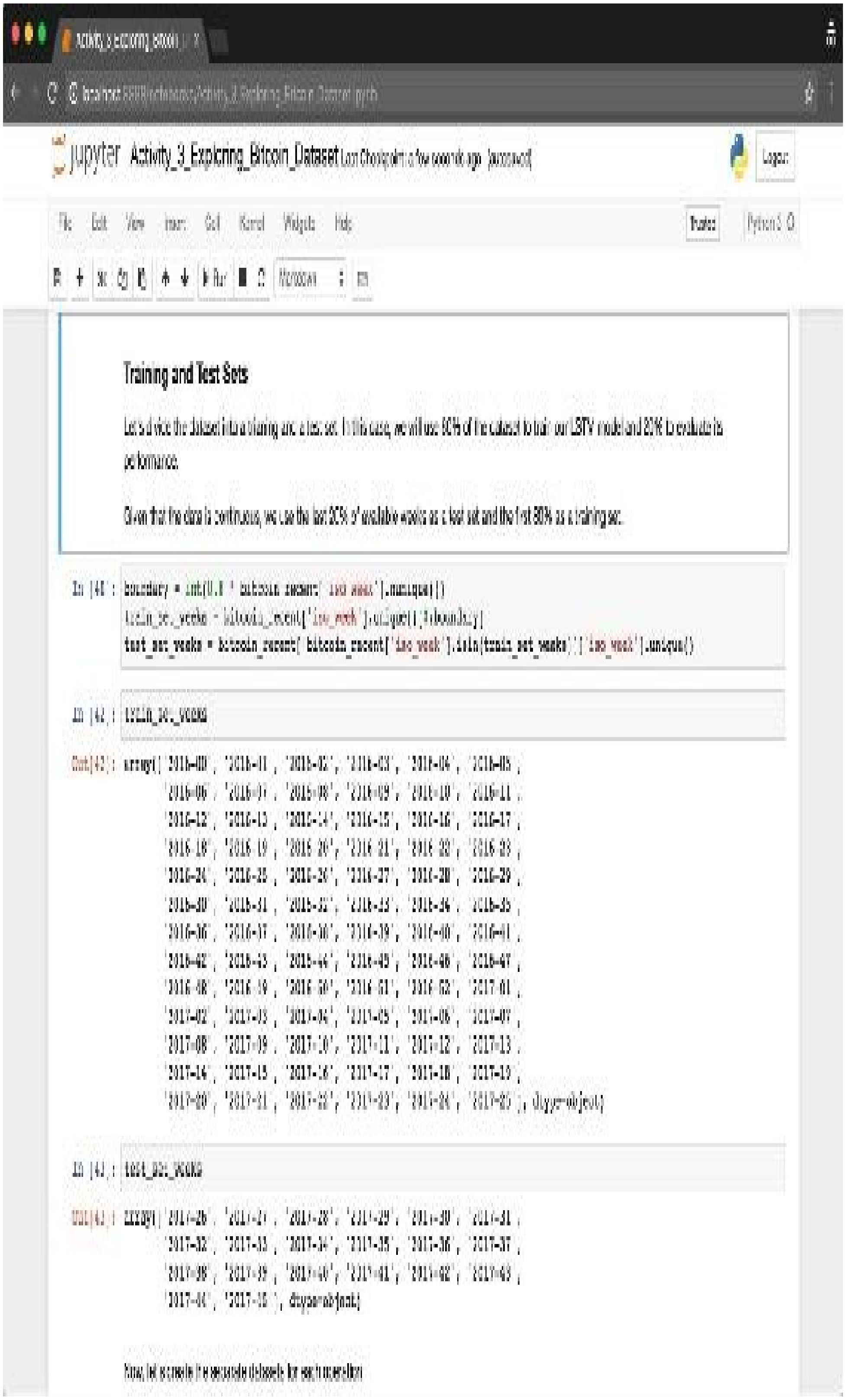


Figure 13: Plot that displays the series from the normalized variable close_point_relative_normalization

In order to evaluate how well our model performs, we need to test its accuracy versus some other data. We do that by creating two datasets: a training set and a test set. In this activity, we will use 80 percent of the dataset to train our LSTM model and 20 percent to evaluate its performance.

Given that the data is continuous and in the form of a time series, we use the last 20 percent of available weeks as a test set and the first 80 percent as a training set:


Activity_3_Exploring_Bitcoin
localhost:8888/notebooks/Activity_3_Exploring_Bitcoin_Dataset.ipynb
jupyter Activity_3_Exploring_Bitcoin_Dataset Last Checkpoint: a few seconds ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Markdown

Training and Test Sets

Let's divide the dataset into a training and a test set. In this case, we will use 80% of the dataset to train our LSTM model and 20% to evaluate its performance.

Given that the data is continuous, we use the last 20% of available weeks as a test set and the first 80% as a training set.

```
In [41]: boundary = int(0.8 * bitcoin_recent['iso_week'].nunique())
         train_set_weeks = bitcoin_recent['iso_week'].unique()[0:boundary]
         test_set_weeks = bitcoin_recent[~bitcoin_recent['iso_week'].isin(train_set_weeks)]['iso_week'].unique()
```

```
In [42]: train_set_weeks
```

```
Out[42]: array(['2016-00', '2016-01', '2016-02', '2016-03', '2016-04', '2016-05',
               '2016-06', '2016-07', '2016-08', '2016-09', '2016-10', '2016-11',
               '2016-12', '2016-13', '2016-14', '2016-15', '2016-16', '2016-17',
               '2016-18', '2016-19', '2016-20', '2016-21', '2016-22', '2016-23',
               '2016-24', '2016-25', '2016-26', '2016-27', '2016-28', '2016-29',
               '2016-30', '2016-31', '2016-32', '2016-33', '2016-34', '2016-35',
               '2016-36', '2016-37', '2016-38', '2016-39', '2016-40', '2016-41',
               '2016-42', '2016-43', '2016-44', '2016-45', '2016-46', '2016-47',
               '2016-48', '2016-49', '2016-50', '2016-51', '2016-52', '2017-01',
               '2017-02', '2017-03', '2017-04', '2017-05', '2017-06', '2017-07',
               '2017-08', '2017-09', '2017-10', '2017-11', '2017-12', '2017-13',
               '2017-14', '2017-15', '2017-16', '2017-17', '2017-18', '2017-19',
               '2017-20', '2017-21', '2017-22', '2017-23', '2017-24', '2017-25'], dtype=object)
```

```
In [43]: test_set_weeks
```

```
Out[43]: array(['2017-26', '2017-27', '2017-28', '2017-29', '2017-30', '2017-31',
               '2017-32', '2017-33', '2017-34', '2017-35', '2017-36', '2017-37',
               '2017-38', '2017-39', '2017-40', '2017-41', '2017-42', '2017-43',
               '2017-44', '2017-45'], dtype=object)
```

Now, let's create the separate datasets for each operation.

Figure 14: Using weeks to create a training and a test set

Finally, navigate to the **Storing Output** section and save the filtered variable to disk, as follows:

```
test_dataset.to_csv('data/test_dataset.csv
', index=False)
train_dataset.to_csv('data/train_dataset.c
sv', index=False)
bitcoin_recent.to_csv('data/bitcoin_recent
.csv', index=False)
```

NOTE

For the reference solution, use the `Code/Lesson-2/activity_3` folder.

In this section, we explored the Bitcoin dataset and prepared it for a deep learning model.

We learned that during the year 2017, the prices of Bitcoin skyrocketed. This phenomenon takes a long time to take place—and may be influenced by a number of external factors that this data alone doesn't explain (for instance, the emergence of other cryptocurrencies). We also used the point-relative normalization technique to process the Bitcoin dataset in weekly chunks. We do this to train an LSTM network to learn the weekly patterns of Bitcoin price changes so that it can predict a full week into the future.

However, Bitcoin statistics show significant fluctuations on a weekly basis. Can we predict the price of Bitcoin in the future? What will those prices be seven days from now? We will be building a deep learning model to explore that question in our next section using Keras.

Using Keras as a TensorFlow Interface

This section focuses on Keras. We are using Keras because it simplifies the TensorFlow interface into general abstractions. In the backend, the computations are still performed in TensorFlow—and the graph is still built using TensorFlow components—but the interface is much simpler. We spend less time worrying about individual components, such as variables and operations, and spend more time building the network as a computational unit. Keras makes it easy to experiment with different architectures and hyperparameters, moving more quickly towards a performant solution.

As of TensorFlow 1.4.0 (November 2017), Keras is now officially distributed with TensorFlow as `tf.keras`. This suggests that Keras is now tightly integrated with TensorFlow and that it will likely continue to be developed as an open source tool for a long period of time.

Model Components

As we have seen in *Lesson 1*, *Introduction to Neural Networks and Deep Learning*, LSTM networks also have input, hidden, and output layers. Each hidden layer has an activation function which evaluates that layer's associated weights and biases. As expected, the network moves data sequentially from one layer to another and evaluates the results by the output at every iteration (that is, an epoch).

Keras provides intuitive classes that represent each one of those components:

Component	**Keras Class**
High-level abstraction of a complete sequential neural network.	`keras.models.Sequential()`
Dense, fully-connected layer.	`keras.layers.core.Dense()`
Activation function.	`keras.layers.core.Activation()`
LSTM recurrent neural network. This class contains components that are exclusive to this architecture, most of which are abstracted by Keras.	`keras.layers.recurrent.LSTM()`

Table 3: Description of key components from the Keras

API. We will be using these components to build a deep learning model.

Keras' `keras.models.Sequential()` component represents a whole sequential neural network. That Python class can be instantiated on its own, then have other components added to it subsequently.

We are interested in building an LSTM network because those networks perform well with sequential data—and time-series is a kind of sequential data. Using Keras, the complete LSTM network would be implemented as follows:

```
from keras.models import Sequential
from keras.layers.recurrent import LSTM
from keras.layers.core import Dense,
Activation

model = Sequential()

model.add(LSTM(
units=number_of_periods,
input_shape=(period_length,
number_of_periods)
return_sequences=False), stateful=True)

model.add(Dense(units=period_length))

model.add(Activation("linear"))
model.compile(loss="mse",
optimizer="rmsprop")
```

Snippet 1: LSTM implementation using Keras

This implementation will be further optimized in *Lesson 3*, *Model Evaluation and Optimization*.

Keras abstraction allows for one to focus on the key elements that make a deep learning system more performant: what the right sequence of components is, how many layers and nodes to include, and which activation function to use. All of these choices are determined by either the order in which components are added to the instantiated `keras.models.Sequential()` class or by parameters passed to each component instantiation (that is, `Activation("linear")`). The final `model.compile()` step builds the neural network using TensorFlow components.

After the network is built, we train our network using the `model.fit()` method. This will yield a trained model that can be used to make predictions:

```
model.fit(
X_train, Y_train,
batch_size=32, epochs=epochs)
```

Snippet 2.1: Usage of `model.fit()`

The variables `X_train` and `Y_train` are, respectively, a set used for training and a smaller set used for evaluating the loss function (that is, testing how well the network predicts data).

Finally, we can make predictions using the `model.predict()` method:

```
model.predict(x=X_train)
```

Snippet 2.2: Usage of `model.predict()`

The previous steps cover the Keras paradigm for working with neural networks. Despite the fact that different architectures can be dealt with in very different ways, Keras simplifies the interface for working with different architectures by using three components - network architecture, fit, and predict:

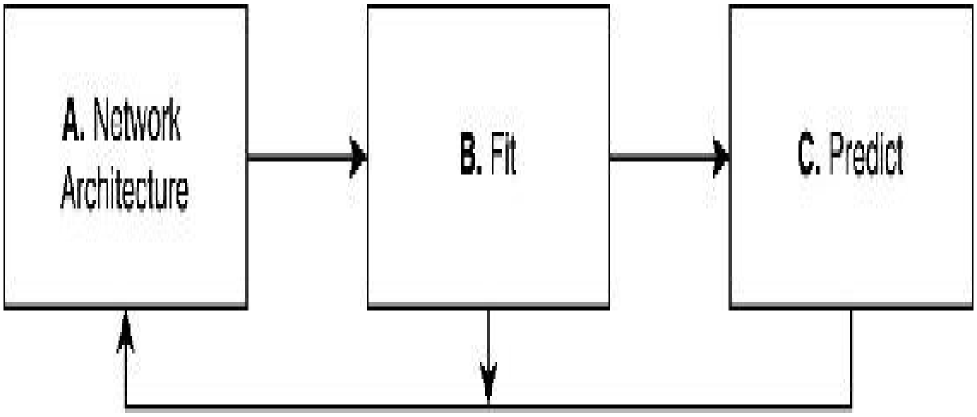


Figure 15: The Keras neural network paradigm: A. design a neural network architecture, B. Train a neural network (or Fit), and C. Make predictions

Keras allows for much greater control within each of those steps. However, its focus is to make it as easy as possible for users to create neural networks in as little time as possible. That means that we can start with a simple model, then add complexity to each one of the steps above to make that initial model perform better.

We will take advantage of that paradigm during our upcoming activity and lessons. In the next activity, we will create the simplest LSTM network possible. Then, in *Lesson 3*, *Model Evaluation and Optimization*, we will continuously evaluate and alter that network to make it more robust and performant.

Activity 4 – Creating a TensorFlow Model Using Keras

In this activity, we will create an LSTM model using Keras.

Keras serves as an interface for lower-level programs; in this case, TensorFlow. When we use Keras to design our neural network, that neural network is *compiled* as a TensorFlow computation graph.

Navigate to the open instance of the Jupyter Notebook `Activity_4_Creating_a_TensorFlow_Model_Using_Keras.ipynb`. Now, execute all cells under the header **Building a Model**. In that section, we build our first LSTM model parametrizing two values: the input size of the training observation (1 equivalent for a single day) and the output size for the predicted period—in our case, seven days:

instance in which we build the first version of our LSTM model

Use the Jupyter Notebook `Activity_4_Creating_a_TensorFlow_Model_Using_Keras.ipynb` to build the same model from the *Model Components* section, parametrizing the period length of input and of output to allow for experimentation.

After the model is compiled, we proceed to storing it as an `h5` file on disk. It is a good practice to store versions of your model on disk occasionally, so that you keep a version of the model architecture alongside its predictive capabilities.

Still on the same Jupyter Notebook, navigate to the header **Saving Model**. In that section, we will store the model as a file on disk with the following command:

```
model.save('bitcoin_lstm_v0.h5')
```

The model `'bitcoin_lstm_v0.h5'` hasn't been trained yet. When saving a model without prior training, one effectively only saves the architecture of the model. That same model can later be loaded by using Keras' `load_model()` function, as follows:

```
model = keras.models.load_model('bitcoin_lstm_v0.h5')
```

NOTE

You may encounter the following warning when loading the Keras library:

```
Using TensorFlow backend.
```

Keras can be configured to use an other backend instead of TensorFlow (that is, Theano). In order to avoid this message, you can create a file called `keras.json` and configure its backend there. The correct configuration of that file depends on your system. Hence, it is recommended that you visit Keras' official documentation on the topic at https://keras.io/backend/.

NOTE

For the reference solution, use the `Code/Lesson-2/activity_4` folder.

In this section, we have learned how to build a deep learning model using Keras, an interface for TensorFlow. We studied core components from Keras and used those components to build the first version of our Bitcoin price-predicting system based on an LSTM model.

In our next section, we will discuss how to put all the components from this lesson together into a (nearly complete) deep learning system. That system will yield our very first predictions, serving as a starting point for future improvements.

From Data Preparation to Modeling

This section focuses on the implementation aspects of a Deep Learning system. We will use the Bitcoin data from*, Choosing the Right Model Architecture* and the Keras knowledge from, *Using Keras as a TensorFlow Interface* to put both of these components together. This section concludes the lesson by building a system that reads data from a disk and feeds it into a model as a single piece of software.

Training a Neural Network

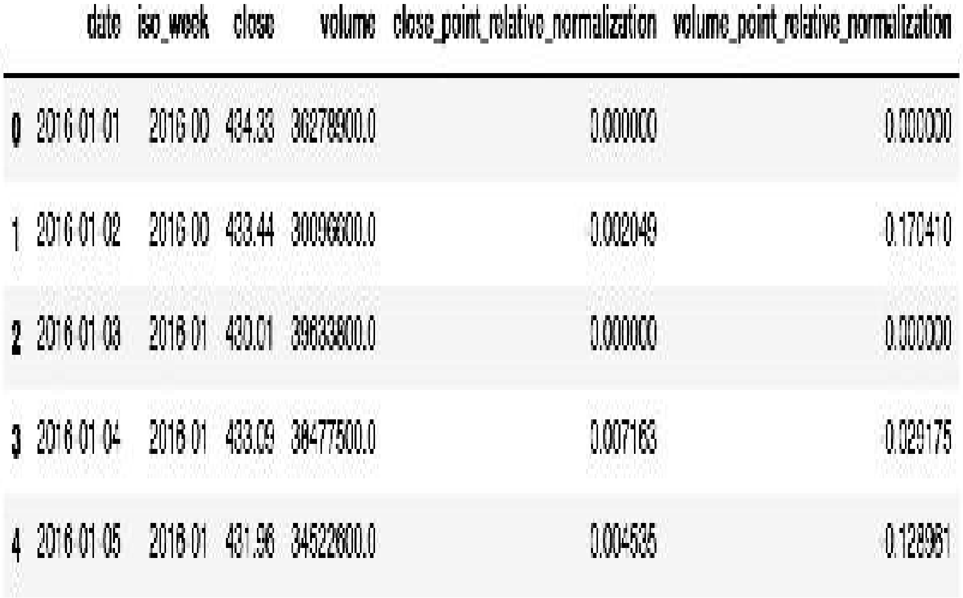
Neural networks can take long periods of time to train. Many factors affect how long that process may take. Among them, three factors are commonly considered the most important:

- The network's architecture
- How many layers and neurons the network has
- How much data there is to be used in the training process

Other factors may also greatly impact how long a network takes to train, but most of the optimization that a neural network can have when addressing a business problem comes from exploring those three.

We will be using the normalized data from our previous section. Recall that we have stored the training data in a file called `train_dataset.csv`. We will load that dataset into memory using `pandas` for easy exploration:

```
import pandas as pd
   train =
pd.read_csv('data/train_dataset.csv')
```



	date	iso_week	close	volume	close_point_relative_normalization	volume_point_relative_normalization
0	2016-01-01	2016-00	434.33	36278900.0	0.000000	0.000000
1	2016-01-02	2016-00	433.44	30096600.0	0.002049	0.170410
2	2016-01-03	2016-01	430.01	39633800.0	0.000000	0.000000
3	2016-01-04	2016-01	433.09	38477500.0	0.007163	0.029175
4	2016-01-05	2016-01	431.96	34522600.0	0.004535	0.128961

Figure 17: Table showing the first five rows of the training dataset loaded from the train_d–ataset.csv file

We will be using the series from the variable `close_point_relative_normalization`, which is a normalized series of the Bitcoin closing prices—from the variable `close`—since the beginning of 2016.

The variable `close_point_relative_normalization` has been normalized on a weekly basis. Each observation from the week's period is made relative to the difference from the closing prices on the first day of the period. This normalization step is important and will help our network train faster.

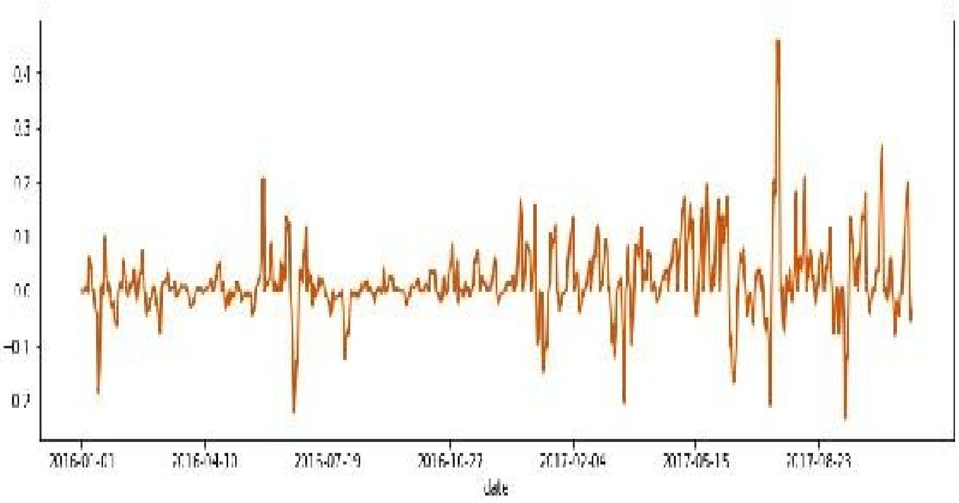


Figure 18: Plot that displays the series from the normalized variable close_point_relative_normalization. This variable will be used to train our LSTM model.

RESHAPING TIME-SERIES DATA

Neural networks typically work with vectors and tensors, both mathematical objects that organize data in a number of dimensions. Each neural network implemented in Keras will have either a vector or a tensor that is organized according to a specification as input. At first, understanding how to reshape the data into the format expected by a given layer can be confusing. To avoid confusion, it is advised to start with a network with as little components as possible, then add components gradually. Keras' official documentation (under the section **Layers**) is essential for learning about the requirements for each kind of layer.

NOTE

The Keras official documentation is available at https://keras.io/layers/core/. That link takes you directly to the **Layers** section.

NOTE

`NumPy` is a popular Python library used for performing numerical computations. It is used by the deep learning community to manipulate vectors and tensors and prepare them for deep learning systems.

In particular, the `numpy.reshape()` method is very important when adapting data for deep learning models. That model allows for the manipulation of `NumPy` arrays, which are Python objects analogous to vectors and tensors.

We now organize the prices from the variable `close_point_relative_normalization` using the weeks of both 2016 and 2017. We create distinct groups containing seven observations each (one for each day of the week) for a total of 77 complete weeks. We do that because we are interested in predicting the prices of a week's worth of trading.

NOTE

We use the ISO standard to determine the beginning and the end of a week. Other kinds of organizations are entirely possible. This one is simple and intuitive to follow, but there is room for improvement.

LSTM networks work with three-dimensional tensors. Each one of those dimensions represents an important property for the network. These dimensions are:

- **Period length**: The period length, that is, how many observations there are on a period

- **Number of periods**: How many periods are available in the dataset
- **Number of features**: Number of features available in the dataset

Our data from the variable `close_point_relative_normalization` is currently a one-dimensional vector—we need to reshape it to match those three dimensions.

We will be using a week's period. Hence, our period length is seven days (period length = 7). We have 77 complete weeks available in our data. We will be using the very last of those weeks to test our model against during its training period. That leaves us with 76 distinct weeks (number of periods = 76). Finally, we will be using a single feature in this network (number of features = 1)—we will include more features in future versions.

How can we reshape the data to match those dimensions? We will be using a combination of base Python properties and the `reshape()` from the `numpy` library. First, we create the 76 distinct week groups with seven days each using pure Python:

```
group_size = 7
samples = list()
for i in range(0, len(data), group_size):
sample = list(data[i:i + group_size])
if len(sample) == group_size:
samples.append(np.array(sample).reshape(group_size, 1).tolist())

data = np.array(samples)
```

Snippet 3: Python code snippet that creates distinct week groups

The resulting variable `data` is a variable that contains all the right dimensions. The Keras LSTM layer expects these dimensions to be organized in a specific order: number of features, number of observations, and period length. Let's reshape our dataset to match that format:

```
X_train = data[:-1,:].reshape(1, 76, 7)
Y_validation = data[-1].reshape(1, 7)
```

Snippet 5: Snippet shows how to train our model

NOTE

Each Keras layer will expect its input to be organized in specific ways. However, Keras will reshape data accordingly in most cases. Always refer to the Keras documentation on layers (https://keras.io/layers/core/) before adding a new layer or if you encounter issues with the shape layers expect.

Snippet 4 also selects the very last week of our set as a validation set (via `data[-1]`). We will be attempting to predict the very last week in our dataset by using the preceding 76 weeks.

The next step is to use those variables to fit our model:

```
model.fit(x=X_train, y=Y_validation,
epochs=100)
```

LSTMs are computationally expensive models. They may take up to five minutes to train with our dataset in a modern computer. Most of that time is spent at the beginning of the computation, when the algorithm creates the full computation graph. The process gains speed after it starts training:

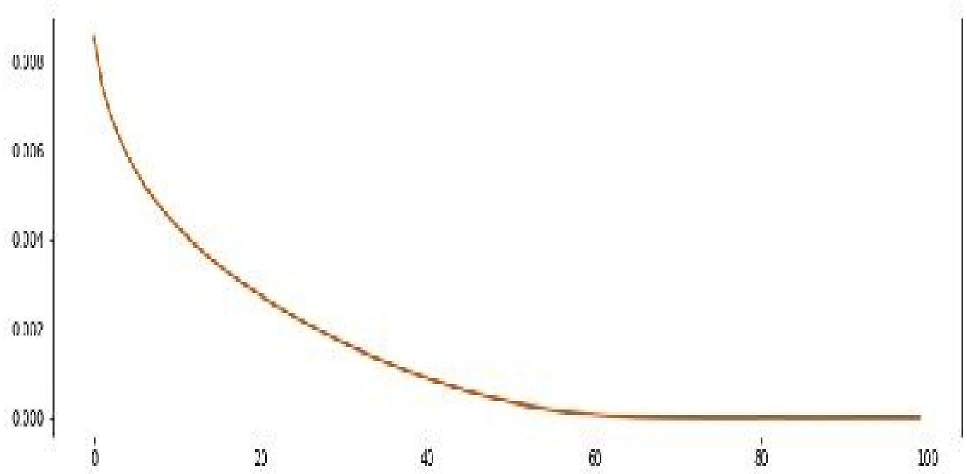


Figure 19: Graph that shows the results of the loss function evaluated at each epoch

NOTE

This compares what the model predicted at each epoch, then compares with the real data using a technique called mean-squared error. This plot shows those results.

At a glance, our network seems to perform very well: it starts with a very small error rate that continuously decreases. Now, what do our predictions tell us?

Making Predictions

After our network has been trained, we can now proceed to making predictions. We will be making predictions for a future week beyond our time period.

Once we have trained our model with `model.fit(),` making predictions is trivial:

```
model.predict(x=X_train)
```

Snippet 6: Making a prediction using the same data that we previously used for training

We use the same data for making predictions as the data used for training (the `X_train` variable). If we have more data available, we can use that instead—given that we reshape it to the format the LSTM requires.

OVERFITTING

When a neural network overfits to a validation set, it means that it learns patterns present in the training set, but is unable to generalize it to unseen data (for instance, the test set). During our next lesson, we will learn how to avoid overfitting and create a system for both evaluating our network and increasing its performance:

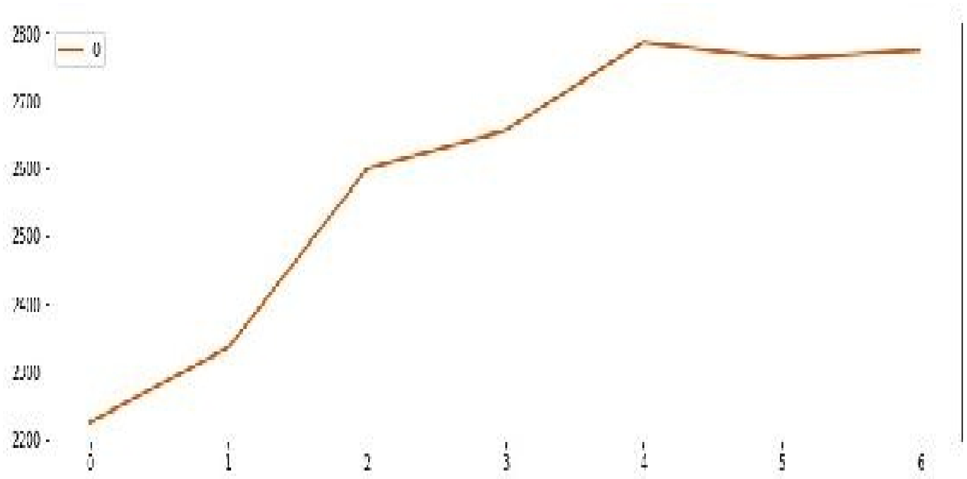


Figure 20: After denormalization, our LSTM model predicted that in late July 2017, the prices of Bitcoin would increase from $2,200 to roughly $2,800, a 30 percent increase in a single week

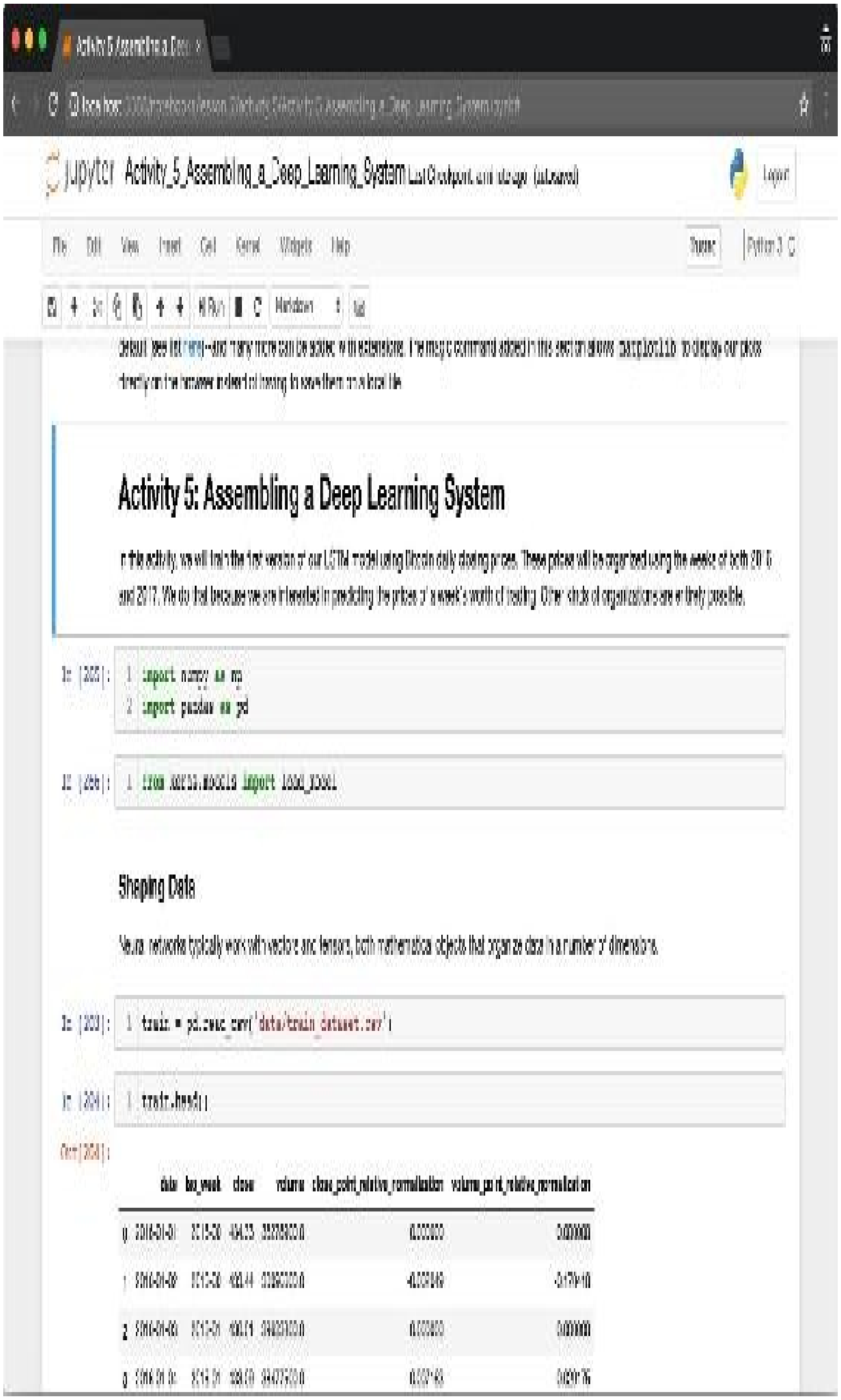
Activity 5 – Assembling a Deep Learning System

In this activity, we bring together all the essential pieces for building a basic deep learning system: data, model, and prediction.

We will continue to use Jupyter Notebooks, and will use the data prepared in previous exercises (`data/train_dataset.csv`) as well as the model that we stored locally (`bitcoin_lstm_v0.h5`).

1. After starting a Jupyter Notebook instance, navigate to the Notebook called `Activity_5_Assembling_a_Deep_Learning_System.ipynb` and open it. Execute the cells from the header to load the required

components and then navigate to the header **Shaping Data**:


Activity_5_Assembling_a_Deep_Learning_System Last Checkpoint: ... (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Markdown

default (see list here)—and many more can be added with extensions. The magic command added in this section allows matplotlib to display our plots directly on the browser instead of having to save them on a local file.

Activity 5: Assembling a Deep Learning System

In this activity, we will train the first version of our LSTM model using Bitcoin daily closing prices. These prices will be organized using the weeks of both 2016 and 2017. We do that because we are interested in predicting the prices of a week's worth of trading. Other kinds of organizations are entirely possible.

```
In [205]:
import numpy as np
import pandas as pd
```

```
In [206]:
from keras.models import load_model
```

Shaping Data

Neural networks typically work with vectors and tensors, both mathematical objects that organize data in a number of dimensions.

```
In [203]:
train = pd.read_csv('data/train_dataset.csv')
```

```
In [204]:
train.head()
```

Out[204]:

	date	iso_week	close	volume	close_point_relative_normalization	volume_point_relative_normalization
0	2016-01-01	2015-52	434.33	36278900.0	0.000000	0.000000
1	2016-01-02	2015-52	433.44	30096600.0	-0.002049	-0.170410
2	2016-01-03	2016-01	430.01	39633800.0	0.000000	0.000000
3	2016-01-04	2016-01	433.09	38477500.0	0.007163	-0.029175

Figure 21: Plot that displays the series from the normalized variable close_point_relative_normalization

NOTE

> The `close_point_relative_normalization` variable will be used to train our LSTM model.

We will start by loading the dataset we prepared during our previous activities. We use `pandas` to load that dataset into memory.

2. Load the training dataset into memory using pandas, as follows:

```
train =
pd.read_csv('data/train_dataset.csv')
```

3. Now, quickly inspect the dataset by executing the following command:

```
train.head()
```

As explained in this lesson, LSTM networks require tensors with three dimensions. These dimensions are: period length, number of periods, and number of features.

Now, proceed to creating weekly groups, then rearrange the resulting array to match those dimensions.

4. Feel free to use the provided function `create_groups()` to perform this operation:

```
create_groups(data=train,
group_size=7)
```

The default values for that function are 7 days. What would happen if you changed that number to a different value, for instance, 10?

Now, make sure to the data into two sets: training and validation. We do this by assigning the last week from the Bitcoin prices dataset to the evaluation set. We then train the network to evaluate that last week.

Separate the last week of the training data and reshape it using `numpy.reshape()`. Reshaping is important, as the LSTM model only accepts data organized in this way:

```
X_train = data[:-1,:].reshape(1, 76,
7)
Y_validation = data[-1].reshape(1, 7)
```

Our data is now ready to be used in training. Now we load our previously saved model and train it with a given number of epochs.

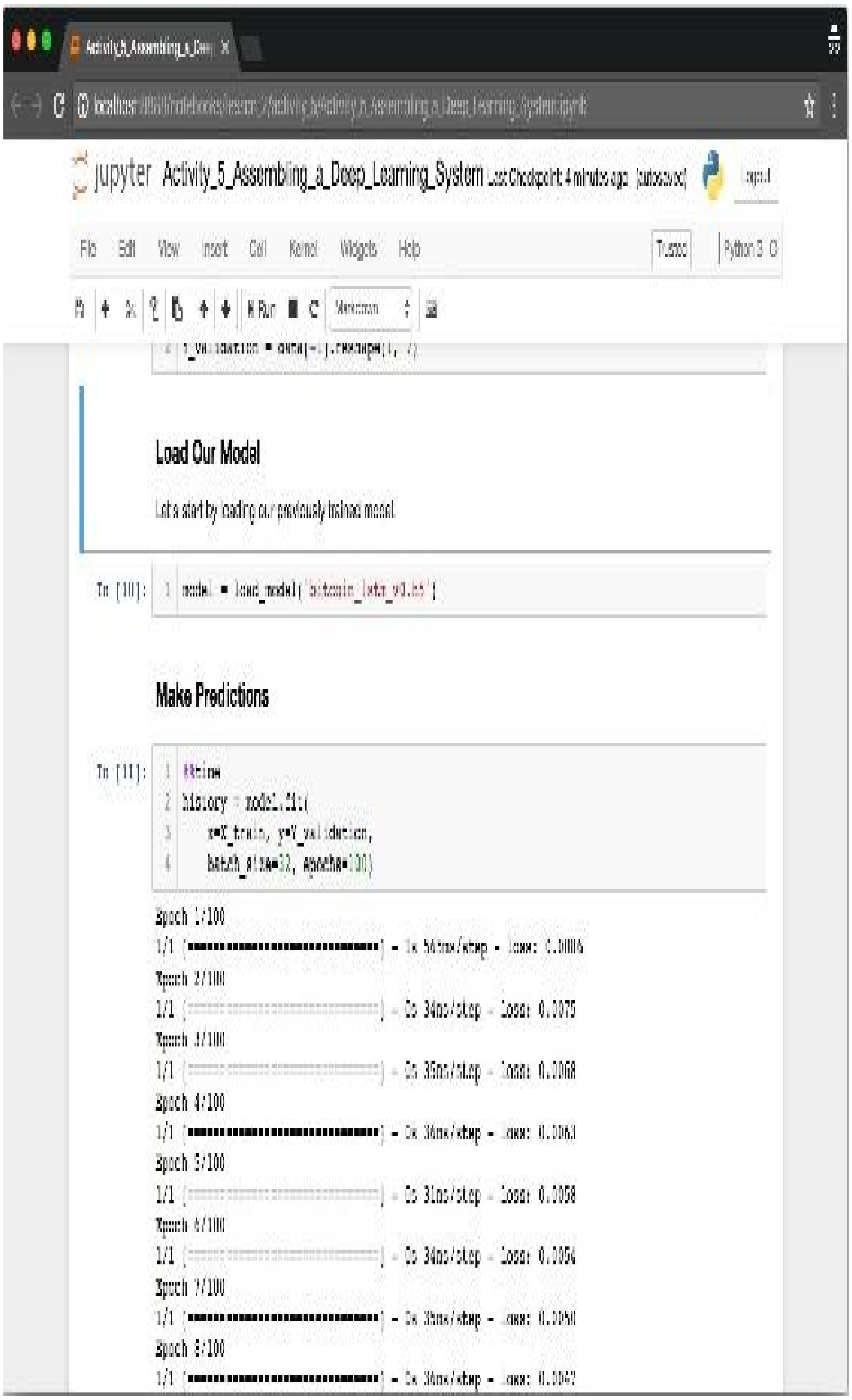
5. Navigate to the header **Load Our Model** and load our previously trained model:

   ```
   model =
   load_model('bitcoin_lstm_v0.h5')
   ```

6. And now, train that model with our training data `X_train` and `Y_validation`:

   ```
   history = model.fit(
   x=X_train, y=Y_validation,
   batch_size=32, epochs=100)
   ```

   Notice that we store the logs of the model in a variable called `history`. The model logs are useful for exploring specific variations in its training accuracy and to understand how well the loss function is performing:


Activity_5_Assembling_a_Deep
localhost:8888/notebooks/lesson_2/activity_5/Activity_5_Assembling_a_Deep_Learning_System.ipynb
jupyter Activity_5_Assembling_a_Deep_Learning_System Last Checkpoint: 4 minutes ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Markdown

```
Y_validation = data[-1:].reshape(1, 7)
```

Load Our Model

Let's start by loading our previously trained model.

```
In [10]: model = load_model('bitcoin_lstm_v0.h5')
```

Make Predictions

```
In [11]: %%time
         history = model.fit(
             x=X_train, y=Y_validation,
             batch_size=32, epochs=100)
```

```
Epoch 1/100
1/1 [==============================] - 1s 56ms/step - loss: 0.0086
Epoch 2/100
1/1 [==============================] - 0s 34ms/step - loss: 0.0075
Epoch 3/100
1/1 [==============================] - 0s 35ms/step - loss: 0.0068
Epoch 4/100
1/1 [==============================] - 0s 36ms/step - loss: 0.0063
Epoch 5/100
1/1 [==============================] - 0s 31ms/step - loss: 0.0058
Epoch 6/100
1/1 [==============================] - 0s 34ms/step - loss: 0.0054
Epoch 7/100
1/1 [==============================] - 0s 35ms/step - loss: 0.0050
Epoch 8/100
1/1 [==============================] - 0s 36ms/step - loss: 0.0047
```

Figure 22: Section of Jupyter Notebook where we load our earlier model and train it with new data

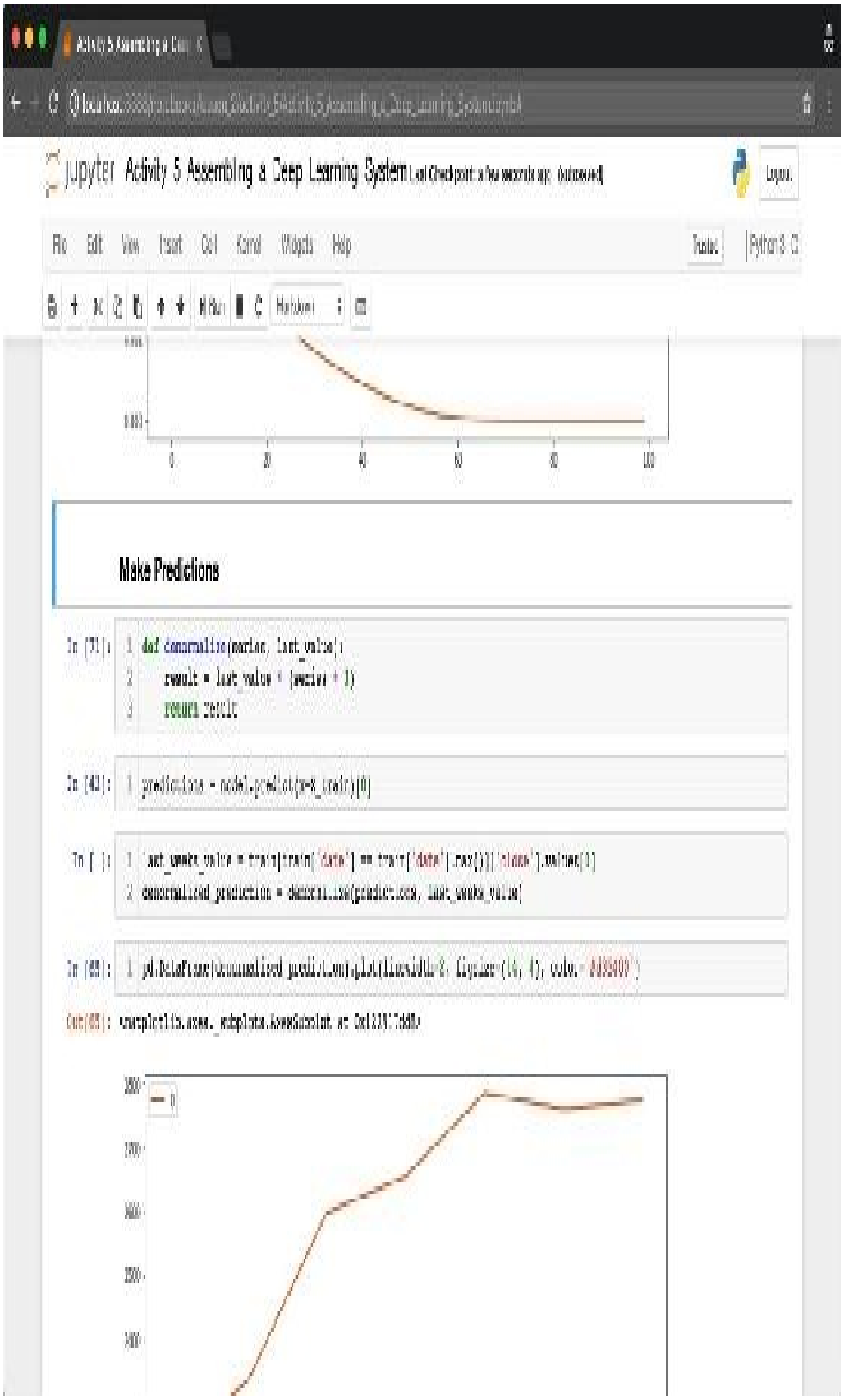
Finally, let's make a prediction with our trained model.

7. Using the same data `X_train`, call the following method:

```
model.predict(x=X_train)
model.predict(x=X_train)
```

8. The model immediately returns a list of normalized values with the prediction for the next seven days. Use the `denormalize()` function to turn the data into US Dollar values. Use the latest values available as a reference for scaling the predicted results:

   ```
   denormalized_prediction =
   denormalize(predictions,
   last_weeks_value)
   ```


Activity 5 Assembling a Deep Learning System Last Checkpoint: a few seconds ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Logout
Markdown
0
20
40
60
80
100
0

Make Predictions

```
In [71]: def denormalize(series, last_value):
             result = last_value * (series + 1)
             return result
```

```
In [43]: predictions = model.predict(x=X_train)[0]
```

```
In [ ]: last_weeks_value = train[train['date'] == train['date'].max()]['close'].values[0]
        denormalized_prediction = denormalize(predictions, last_weeks_value)
```

```
In [65]: pd.DataFrame(denormalized_prediction).plot(linewidth=2, figsize=(6, 4), color='#d35400')
```

```
Out[65]: <matplotlib.axes._subplots.AxesSubplot at 0x1231f7dd8>
```

Figure 23: Section of Jupyter Notebook where we predict the prices of Bitcoin for the next seven days. Our predictions suggest a great price surge of about 30 percent.

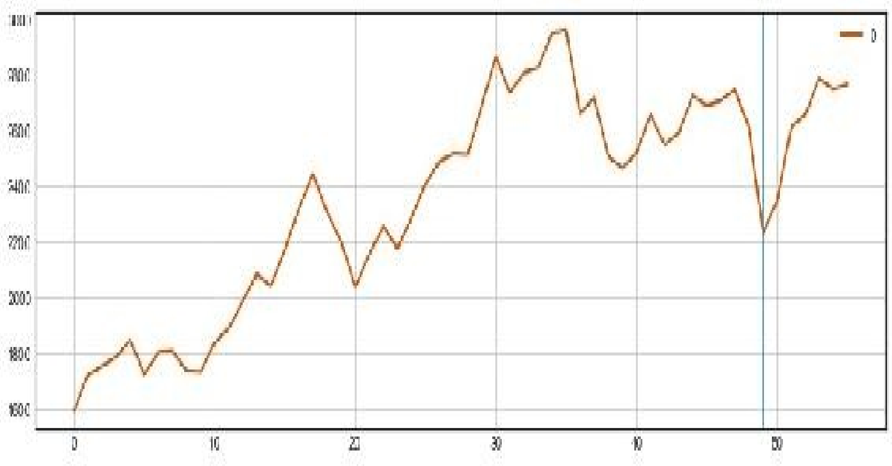


Figure 24: Projection of Bitcoin prices for seven days in the future using the LSTM model we just built

> **NOTE**
>
> We combine both time-series in this graph: the real data (before the line) and the predicted data (after the line). The model shows variance similar to the patterns seen before and it suggests a price increase during the following seven days period.

9. After you are done experimenting, save your model with the following command:

```
model.save('bitcoin_lstm_v0_trained.h5
')
```

We will save this trained network for future reference and compare its performance with other models.

The network may have learned patterns from our data, but how can it do that with such a simple architecture and so little data? LSTMs are powerful tools for learning patterns from data. However, we will learn in our next sessions that they can also suffer from *overfitting*, a phenomenon common in neural networks in which they learn patterns from the training data that are useless when predicting real-world patterns. We will learn how to deal with that and how to improve our network to make useful predictions.

> **NOTE**
>
> For the reference solution, use the `Code/Lesson-2/activity_5` folder.

Summary

In this lesson, we have assembled a complete deep learning system: from data to prediction. The model created in this activity needs a number of improvements before it can be considered useful. However, it serves as a great starting point from which we will continuously improve.

Our next lesson will explore techniques for measuring the performance of our model and will continue to make modifications until we reach a model that is both useful and robust.

Chapter 3. Model Evaluation and Optimization

This lesson focuses on how to evaluate a neural network model. Different than working with other kinds of models, when working with neural networks, we modify the network's *hyperparameters* to improve its performance. However, before altering any parameters, we need to measure how the model performs.

Lesson Objectives

In this lesson, you will:

- Evaluate a model
  - Explore the types of problems addressed by neural networks
  - Explore loss functions, accuracy, and error rates
  - Use TensorBoard
  - Evaluate metrics and techniques
- Hyperparameter optimization
  - Add layers and nodes
  - Explore and add epochs
  - Implement activation functions
  - Use regularization strategies

Model Evaluation

In machine learning, it is common to define two distinct terms: **parameter** and **hyperparameter**. Parameters are properties that affect how a model makes predictions from data. Hyperparameters refer to how a model learns from data. Parameters can be learned from the data and modified dynamically. Hyperparameters are higher-level properties and are not typically learned from data. For a more detailed overview, refer to the book *Python Machine Learning,* by Sebastian Raschka and Vahid Mirjalili (Packt, 2017).

Problem Categories

Generally, there are two categories of problems solved by neural networks: classification and regression. Classification problems regard the prediction of the right categories from data; for instance, if the temperature is *hot* or *cold*. Regression problems are about the prediction of values in a continuous scalar; for instance, what the actual temperature value is?

Problems in these two categories are characterized by the following properties:

- **Classification**: Problems that are characterized by categories. The categories can be different, or not; they can also be about a binary problem. However, they must be clearly assigned to each data element. An example of a classification problem would be to assign the label *car* or *not car* to an image using a Convolutional Neural Network. The MNIST example explored in *Lesson 1*, *Introduction to Neural Networks and Deep Learning,* is another example of a classification problem.

- **Regression**: Problems that are characterized by a continuous variable (that is, a scalar). These problems are measured in terms of ranges, and their evaluations regard how close to the real values the network is. An example is a time-series classification problem in which a Recurrent Neural Network is used to predict the future temperature values. The Bitcoin price-prediction problem is another example of a regression problem.

While the overall structure of how to evaluate these models is the same for both of these problem categories, we employ different techniques for evaluating how models perform. In the following section, we explore these techniques for either classification or regression problems.

> **NOTE**
>
> All of the code snippets in this lesson are implemented in *Activities 6 and 7*. Feel free to follow along, but don't feel that it is mandatory, given that they will be repeated in more detail during the activities.

Loss Functions, Accuracy, and Error Rates

Neural networks utilize functions that measure how the networks perform when compared to a validation set—that is, a part of the data separated to be used as part of the training process. These functions are called **loss functions**.

Loss functions evaluate how *wrong* a neural network's predictions are; then they will propagate those errors back and make adjustments to the network, modifying how individual neurons are activated. Loss functions are key components of neural networks, and choosing the right loss function can have a significant impact on how the network performs.

How are errors propagated to each neuron in a network?

Errors are propagated via a process called backpropagation. Backpropagation is a technique for propagating the errors returned by the loss function back to each neuron in a neural network. Propagated errors affect how neurons activate, and ultimately, how they influence the output of that network.

Many neural network packages, including Keras, use this technique by default.

NOTE

For more information about the mathematics of backpropagation, please refer to *Deep Learning* by Ian Goodfellow et. al., MIT Press, 2016.

We use different loss functions for regression and classification problems. For classification problems, we use accuracy functions (that is, the proportion of times the predictions were correct). While for regression problems, we use error rates (that is, how close the predicted values were to the observed ones).

The following table provides a summary of common loss functions to utilize, alongside their common applications:

Problem Type	**Loss Function**	**Problem**	**Example**
Regression	Mean Squared Error (**MSE**)	Predicting a continuous function. That is, predicting a value within a range of values.	Predicting the temperature in the future using temperature measurements from the past.
Regression	Root Mean Squared Error (**RMSE**)	Same as preceding, but deals with negative values. RMSE typically provides more interpretable results.	Same as preceding.
Regression	Mean Absolute Percentage Error (**MAPE**)	Prediction continuous functions. Has better performance when working with de-normalized ranges.	Predicting the sales for a product using the product properties (for example, price, type, target audience, market conditions).
Classification	Binary Cross-entropy	Classification between two categories or between two values (that is, `true` or `false`).	Predicting if the visitor of a website is male or female based on their browser activity.

Classification	Categorical Cross-entropy	Classification between many categories from a known set of categories.	Predicting the nationality of a speaker based on their accent when speaking a sentence in English.

Table 1: Common loss functions used for classification and regression problems

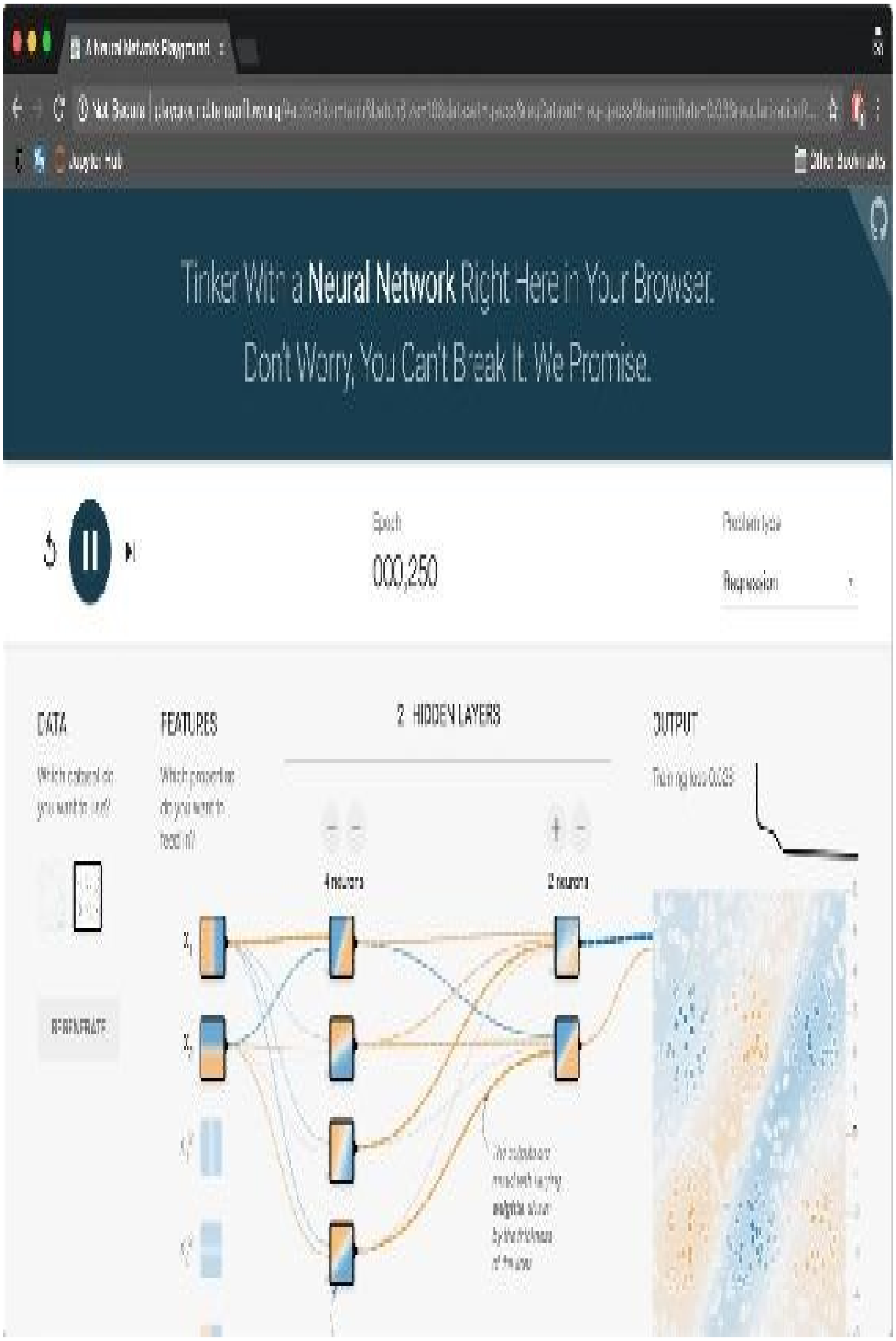
For regression problems, the MSE function is the most common choice. While for classification problems, Binary Cross-entropy (for binary category problems) and Categorical Cross-entropy (for multi-category problems) are common choices. It is advised to start with these loss functions, then experiment with other functions as you evolve your neural network, aiming to gain performance.

The network we develop in *Lesson 2*, *Model Architecture,* uses the MSE as its loss function. In the following section, we explore how that function performs as the network trains.

DIFFERENT LOSS FUNCTIONS, SAME ARCHITECTURE

Before moving ahead to the next section, let's explore, in practical terms, how these problems are different in the context of neural networks.

The TensorFlow Playground application is made available by the TensorFlow team to help us understand how neural networks work. Here, we see a neural network represented with its layers: input (on the left), hidden layers (in the middle), and output (on the right). We can also choose different sample datasets to experiment with on the far-left side. And, finally, on the far-right side, we see the output of the network.



Tinker With a **Neural Network** Right Here in Your Browser.

Don't Worry, You Can't Break It. We Promise.

Epoch 000,250

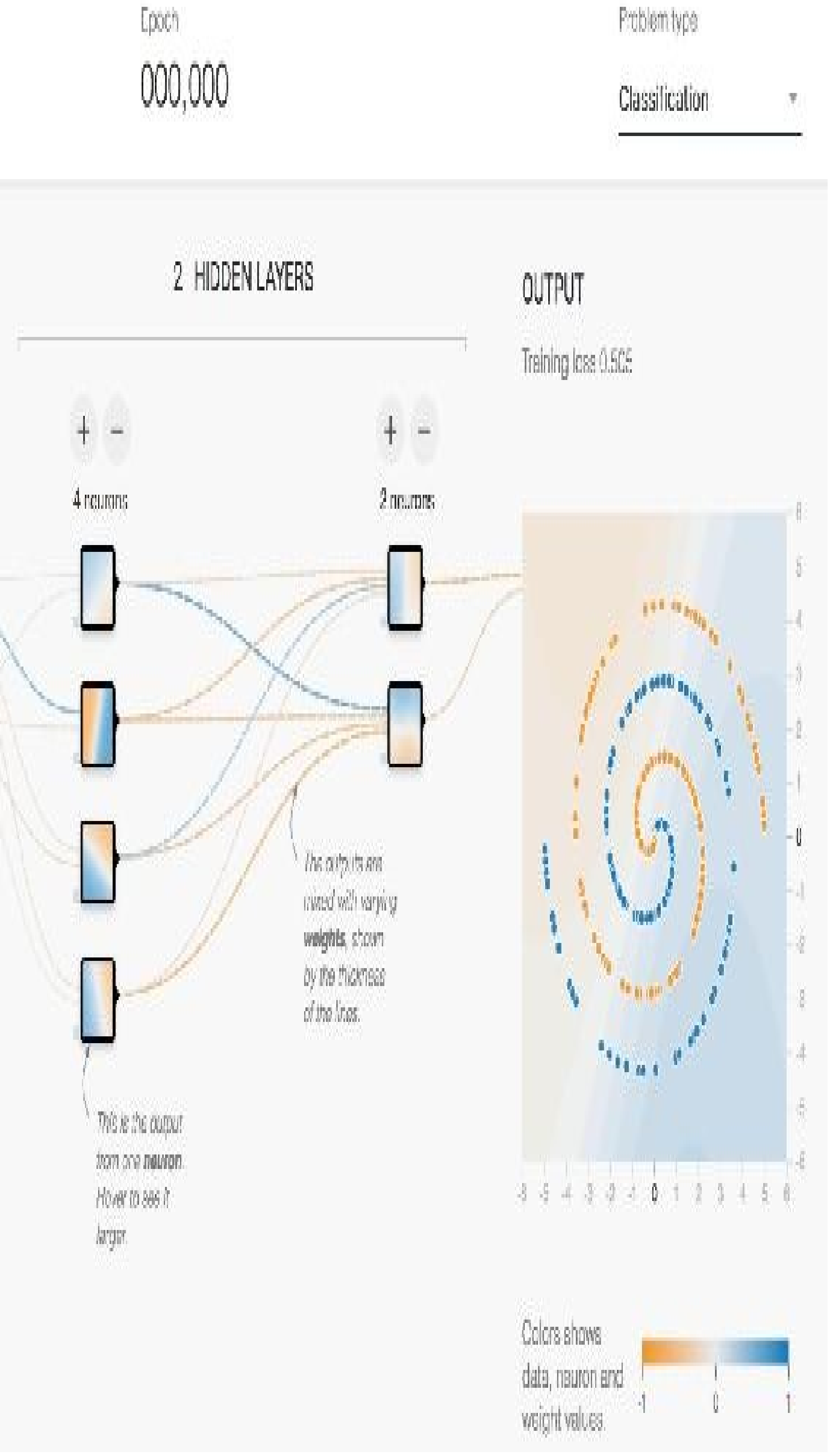
Problem type Regression

A Neural Network Playground
Not Secure | playground.tensorflow.org
Jupyter Hub
Other Bookmarks
DATA
Which dataset do you want to use?
REGENERATE
FEATURES
Which properties do you want to feed in?
2 HIDDEN LAYERS
4 neurons
2 neurons
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Training loss 0.03

Figure 1: TensorFlow Playground web application. Take the parameters for a neural network in this visualization to gain some intuition on how each parameter

affects the model results.

This application helps us explore the different problem categories we discussed in our previous section. When we choose Classification as the Problem type (upper right-hand corner), the dots in the dataset are colored with only two color values: either blue or orange. When we choose Regression, the colors of the dots are colored in a range of color values between orange and blue. When working on classification problems, the network evaluates its loss function based on how many blues and oranges the network has gotten wrong; and when working on classification problems, it checks how far away to the right color values for each dot the network was, as shown in the following image:



Epoch

000,000

Problem type

Classification

2 HIDDEN LAYERS

OUTPUT

Training loss 0.505

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

This is the output from one **neuron**. Hover to see it larger.

Colors shows data, neuron and weight values.

Figure 2: Detail of the TensorFlow Playground application. Different color values are assigned to the dots, depending on the problem type.

After clicking on the play button, we notice that the numbers in the **Training loss** area keep going down as the network continuously trains. The numbers are very similar in each problem category because the loss functions play the same role in both neural networks. However, the actual loss function used for each category is different, and is chosen depending on the problem type.

USING TENSORBOARD

Evaluating neural networks is where TensorBoard excels. As explained in *Lesson 1*, *Introduction to Neural Networks and Deep Learning*, TensorBoard is a suite of visualization tools shipped with TensorFlow. Among other things, one can explore the results of loss function evaluations after each epoch. A great feature of TensorBoard is that one can organize the results of each run separately and compare the resulting loss function metrics for each run. One can then make a decision on which hyperparameters to tune and have a general sense of how the network is performing. The best part is that it is all done in real time.

In order to use TensorBoard with our model, we will use a Keras callback function. We do that by importing the

TensorBoard callback and passing it to our model when calling its fit() function. The following code shows an example of how it would be implemented in the Bitcoin model created in our previous lessons:

```
from keras.callbacks import TensorBoard
 model_name = 'bitcoin_lstm_v0_run_0'
 tensorboard =
TensorBoard(log_dir='./logs/{}'.format(mod
el_name))
 model.fit(x=X_train, y=Y_validate,
 batch_size=1, epochs=100,
 verbose=0, callbacks=[tensorboard])
```

Snippet 1: Snippet that implements a TensorBoard callback in our LSTM model

Keras callback functions are called at the end of each epoch run. In this case, Keras calls the TensorBoard callback to store the results from each run on the disk. There are many other useful callback functions available, and one can create custom ones using the Keras API.

NOTE

Please refer to the Keras callback documentation (https://keras.io/callbacks/) for more information.

After implementing the TensorBoard callback, the `loss` function metrics are now available in the TensorBoard interface. You can now run a TensorBoard process (with `tensorboard --logdir=./logs`) and leave it running while you train your network with `fit()`. The main graphic to evaluate is typically called *loss*. One can add more metrics by passing known metrics to the `metrics` parameter in the `fit()` function; these will then be available for visualization in TensorBoard, but will not be used to adjust the network weights. The interactive graphics will continue to update in real time, which allows you to understand what is happening on every epoch.

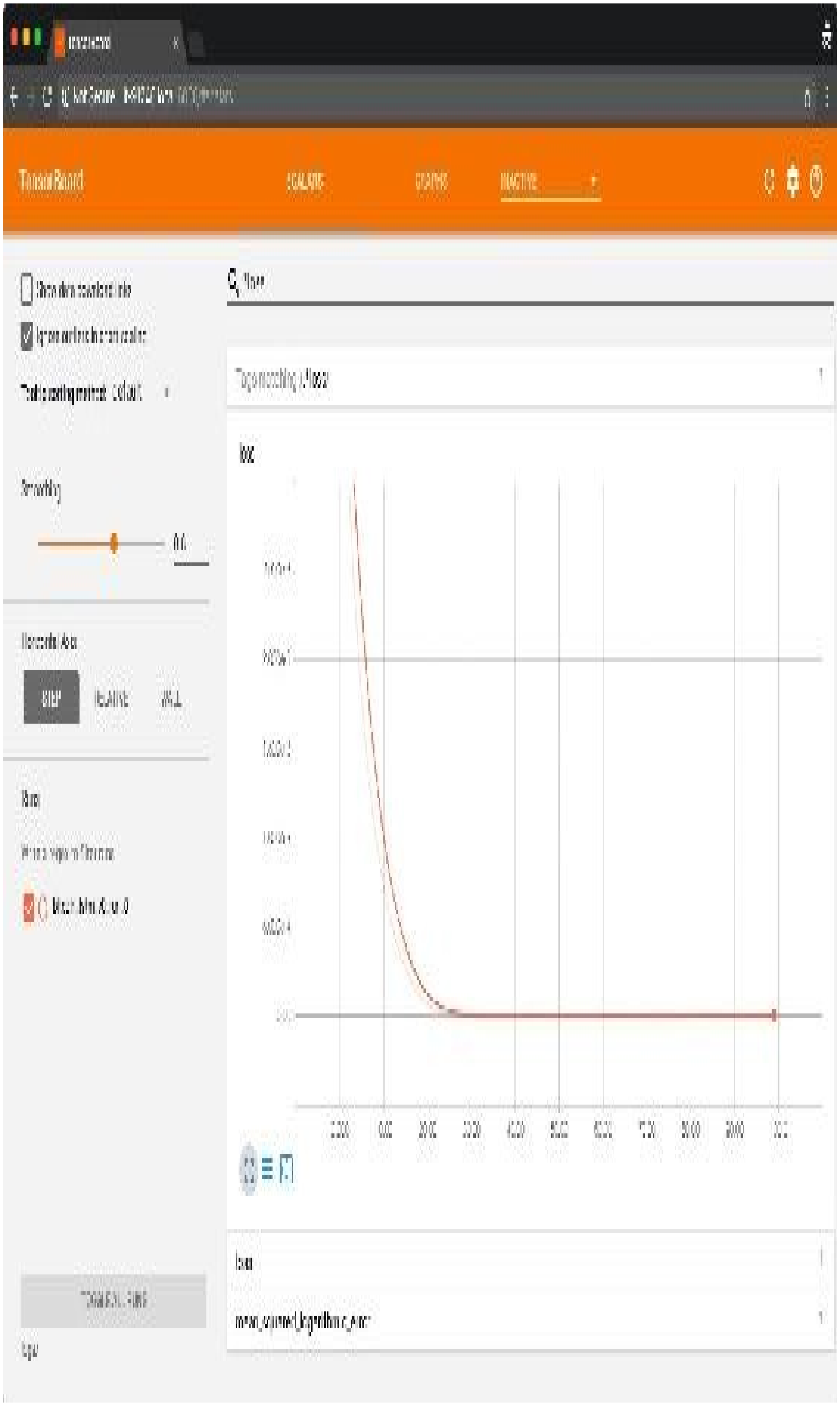


Figure 3: Screenshot of a TensorBoard

instance showing the loss function results alongside other metrics added to the metrics parameter

Implementing Model Evaluation Metrics

In both regression and classification problems, we split the input dataset into three other datasets: train, validation, and test. Both the train and the validation sets are used to train the network. The train set is used by the network as an input, and the validation set is used by the loss function to compare the output of the neural network to the real data, computing how wrong the predictions are. Finally, the test set is used after the network has been trained to measure how the network can perform on data it has never seen before.

> **NOTE**
>
> There isn't a clear rule for determining how the train, validation, and test datasets must be divided. It is a common approach to divide the original dataset as 80 percent train and 20 percent test, then to further divide the train dataset into 80 percent train and 20 percent validation. For more information about this problem, please refer to the book *Python Machine Learning,* by Sebastian Raschka and Vahid Mirjalili (Packt, 2017).

In classification problems, you pass both the data and the labels to the neural network as related but distinct data. The network then learns how data is related to each label. In regression problems, instead of passing data and labels, one passes the variable of interest as one parameter and the variables used for learning patterns as another. Keras provides an interface for both of those use cases with the `fit()` method. See *Snippet 2* for an example:

```
model.fit(x=X_train, y=Y_train,
             batch_size=1, epochs=100,
             verbose=0,
callbacks=[tensorboard],
             validation_split=0.1,

validation_data=(X_validation,
Y_validation))
 Snippet 2: Snippet that illustrates how
to use the validation_split and
validation_data parameters
```

Snippet 2: Snippet that illustrates how to use the `validation_split` *and* `validation_data` *parameters*

> **NOTE**
>
> The `fit()` method can use either the `validation_split` or the `validation_data` parameter, but not both at the same time.

Loss functions evaluate the progress of models and adjust their weights on every run. However, loss functions only describe the relationship between training data and validation data. In order to evaluate if a model is performing correctly, we typically use a third set of data—which is not used to train the network—and compare the predictions made by our model to the values available in that set of data. That is the role of the test set.

Keras provides the method `model.evaluate(),` which makes the process of evaluating a trained neural network against a test set easy. See the following code for an example:

```
model.evaluate(x=X_test, y=Y_test)
```

Snippet 3: Snippet that illustrates how to use the `evaluate()` *method*

The `evaluate()` method returns both the results of the loss function and the results of the functions passed to the `metrics` parameter. We will be using that function frequently in the Bitcoin problem to test how the model performs on the test set.

You will notice that the Bitcoin model looks a bit different than the example above. That is because we are using an LSTM architecture. LSTMs are designed to predict sequences. Because of that, we do not use a set of variables to predict a different single variable—even if it is a regression problem. Instead, we use previous observations from a single variable (or set of variables) to predict future observations of that same variable (or set). The `y` parameter on `Keras.fit()` contains the same variable as the `x` parameter, but only the predicted sequences.

Evaluating the Bitcoin Model

We created a test set during our activities in *Lesson 1*, *Introduction to Neural Networks and Deep Learning*. That test set has 19 weeks of Bitcoin daily price observations, which is equivalent to about 20 percent of the original dataset.

We have also trained our neural network using the other 80 percent of data (that is, the train set with 56 weeks of data, minus one for the validation set) in *Lesson 2*, *Model Architecture,* and stored the trained network on disk (`bitcoin_lstm_v0).` We can now use the `evaluate()` method in each one of the 19 weeks of data from the test set and inspect how that first neural network performs.

In order to do that, though, we have to provide 76 preceding weeks. We have to do this because our network has been trained to predict one week of data using exactly 76 weeks of continuous data (we will deal with this behavior by re-training our network periodically with larger periods in *Lesson 4*, *Productization,* when we deploy a neural network as a web application):

```
combined_set = np.concatenate((train_data,
test_data), axis=1)
```

```
evaluated_weeks = []
  for i in range(0,
validation_data.shape[1]):
  input_series = combined_set[0:,i:i+77]

     X_test =
input_series[0:,:-1].reshape(1,
input_series.shape[1] - 1, )
     Y_test = input_series[0:,-1:][0]

     result = B.model.evaluate(x=X_test,
y=Y_test, verbose=0)
```

```
    y=Y_test, verbose=0)
        evaluated_weeks.append(result)
```

Snippet 4: Snippet that implements the `evaluate()` *method to evaluate the performance of our model in a test dataset*

In the preceding code, we evaluate each week using Keras' `model.evaluate()`, then store its output in the variable `evaluated_weeks`. We then plot the resulting MSE for each week in the following figure:

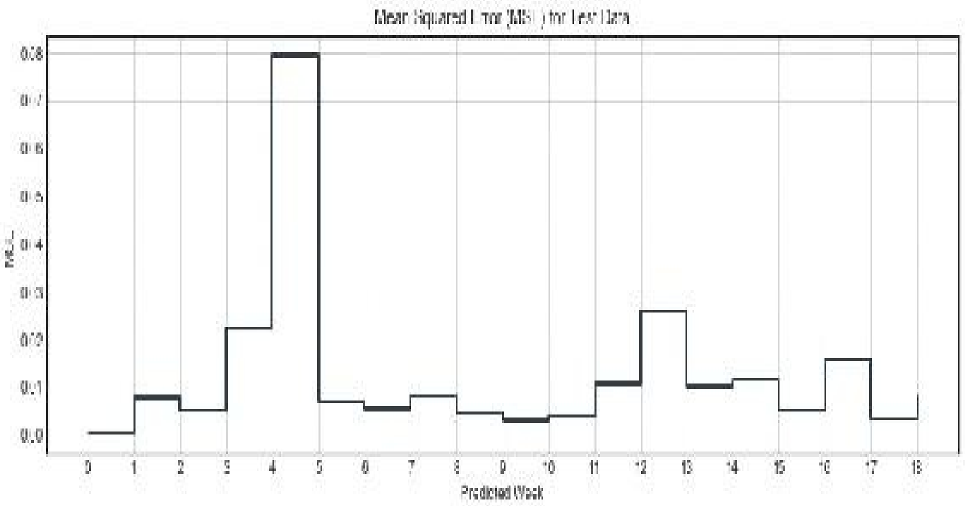


Figure 4: MSE for each week in the test set; notice that in week 5, the model predictions are worse than in any other week

The resulting MSE from our model suggests that our model performs well during most weeks, except for week 5, when its value increases to about `0.08`. Our model seems to be performing well for almost all of the other test weeks.

OVERFITTING

Our first trained network (`bitcoin_lstm_v0`) may be suffering from a phenomenon known as *overfitting*. Overfitting is when a model is trained to optimize a validation set, but it does so at the expense of more generalizable patterns from the phenomenon we are interested in predicting. The main issue with overfitting is that a model learns how to predict the validation set, but fails to predict new data.

The loss function used in our model reaches very low levels (about 2.9 * 10-6) at the end of our training process. Not only that, but this happens early: the MSE loss function used to predict the last week in our data decreases to a stable plateau in about epoch 30. This means that our model is predicting the data from week 77 almost perfectly, using the preceding 76 weeks. Could this be the result of overfitting?

Let's look at Figure 4 again. We know that our LSTM model reaches extremely low values in our validation set (about 2.9 * 10-6), yet it also reaches low values in our test set. The key difference, however, is in the scale. The MSE for each week in our test set is about 4,000 times bigger (on average) than in the test set. This means that the model is performing much worse in our test data than in the validation set. This is worth considering.

The scale, though, hides the power of our LSTM model: even performing much worse in our test set, the predictions' MSE errors are still very, very low. That suggests that our model may be learning patterns from the data.

MODEL PREDICTIONS

One thing is to measure our model comparing MSE errors, and another is to be able to interpret its results intuitively.

Using the same model, let's now create a series of predictions for the following weeks, using 76 weeks as input. We do that by sliding a window of 76 weeks over the complete series (that is, train plus test sets), and making predictions for each of those windows. Predictions are done using the `Keras model.predict()` method:

```
combined_set = np.concatenate((train_data,
test_data), axis=1)
```

```
predicted_weeks = []
 for i in range(0,
validation_data.shape[1] + 1):
 input_series = combined_set[0:,i:i+76]

predicted_weeks.append(B.predict(input_ser
ies))
```

Snippet 5: Snippet that uses the model.predict() method for making predictions for all the weeks of the test dataset

In the preceding code, we make predictions using `model.predict(),` then store these predictions in the `predicted_weeks` variable. We then plot the resulting predictions, making the following figure:

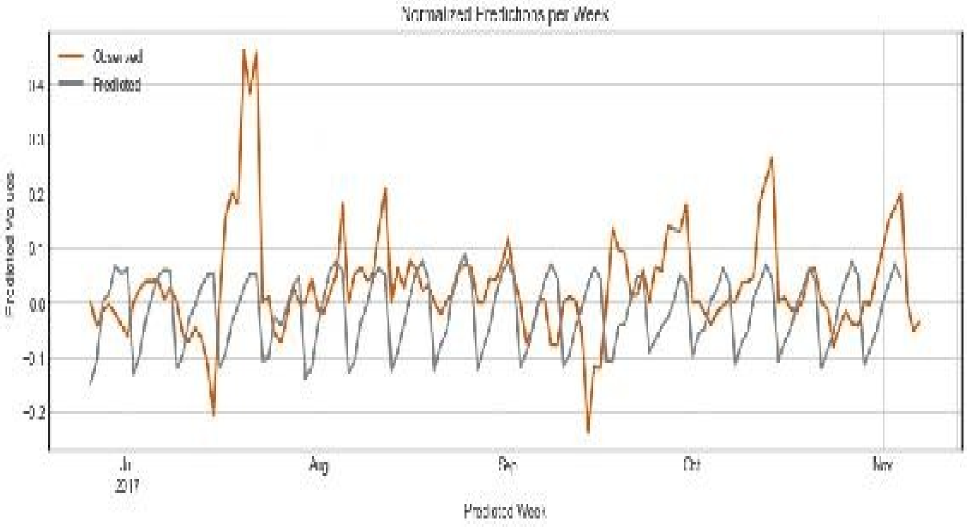


Figure 5: MSE for each week in the test set. Notice that in week 5, the model predictions are worse than in any other week.

The results of our model (as shown in *Figure 5*) suggest that its performance isn't all that bad. By observing the pattern from the *Predicted* line, one can notice that the network has identified a fluctuating pattern happening on a weekly basis, in which the normalized prices go up in the middle of the week, then down by the end of it. With the exception of a few weeks—most notably week 5, the same from our previous MSE analysis—most weeks fall close to the correct values.

Let's now denormalize the predictions so that we can investigate the prediction values using the same scale as the original data (that is, US Dollars). We can do this by implementing a denormalization function that uses the day index from the predicted data to identify the equivalent week on the test data. After that week is identified, the function then takes the first value of that week and uses that value to denormalize the predicted values by using the same point-relative normalization technique, but inverted:

```
def denormalize(reference, series,
```

```
normalized_variable='close_point_relative_
normalization',
 denormalized_variable='close'):
 week_values =
observed[reference['iso_week'] ==
series['iso_week']. values[0]]
 last_value =
week_values[denormalized_variable].values[
0]
 series[denormalized_variable] =
last_value * (series[normalized_variable]
+ 1)

 return series

 predicted_close =
predicted.groupby('iso_week').apply(
 lambda x: denormalize(observed, x))
```

Snippet 6: Denormalization of data using an inverted point-relative normalization technique. The denormalize() function takes the first closing price from the test's first day of an equivalent week.

Our results now compare the predicted values with the test set, using US Dollars. As seen in *Figure 5*, the `bitcoin_lstm_v0` model seems to perform quite well in predicting the Bitcoin prices for the following seven days. But, how can we measure that performance in interpretable terms?

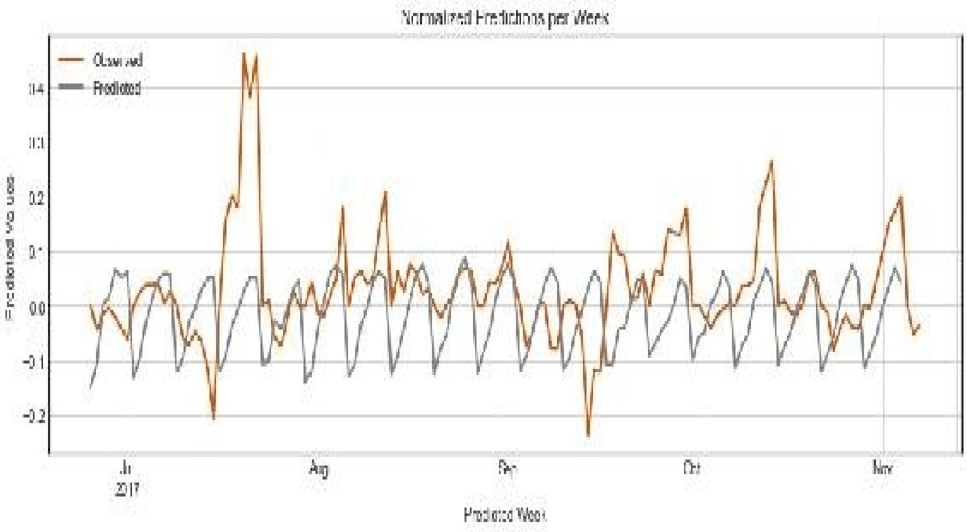


Figure 6: MSE for each week in the test set; notice that in week 5, the model predictions are worse than in any other week

Interpreting Predictions

Our last step is to add interpretability to our predictions. Figure 6 seems to show that our model prediction matches the test data somewhat closely, but how closely?

Keras' `model.evaluate()` function is useful for understanding how a model is performing at each evaluation step. However, given that we are typically using normalized datasets to train neural networks, the metrics generated by the `model.evaluate()` method are also hard to interpret.

In order to solve that problem, we can collect the complete set of predictions from our model and compare it with the test set using two other functions from *Table 1* that are easier to interpret: MAPE and RMSE, implemented as `mape()` and `rmse()`, respectively:

```
def mape(A, B):
return np.mean(np.abs((A - B) / A)) * 100

  def rmse(A, B):
  return np.sqrt(np.square(np.subtract(A,
B)).mean())
```

Snippet 7: Implementation of the `mape()` and `rmse()` functions

NOTE

These functions are implemented using NumPy. Original implementations come from https://stats.stackexchange.com/questions/58391/mean-absolute-percentage-error-mape-in-scikit-learn (MAPE) and https://stackoverflow.com/questions/16774849/mean-squared-error-in-numpy (RMSE).

After comparing our test set with our predictions using both of those functions, we have the following results:

- Denormalized **RMSE**: $399.6
- Denormalized **MAPE**: 8.4 percent

This indicates that our predictions differ, on average, about $399 from real data. That represents a difference of about 8.4 percent from real Bitcoin prices.

These results facilitate the understanding of our predictions. We will continue to use the `model.evaluate()` method to keep track of how our LSTM model is improving, but will also compute both `rmse()` and `mape()` on the complete series on every version of our model to interpret how close we are to predicting Bitcoin prices.

Activity 6 – Creating an Active Training Environment

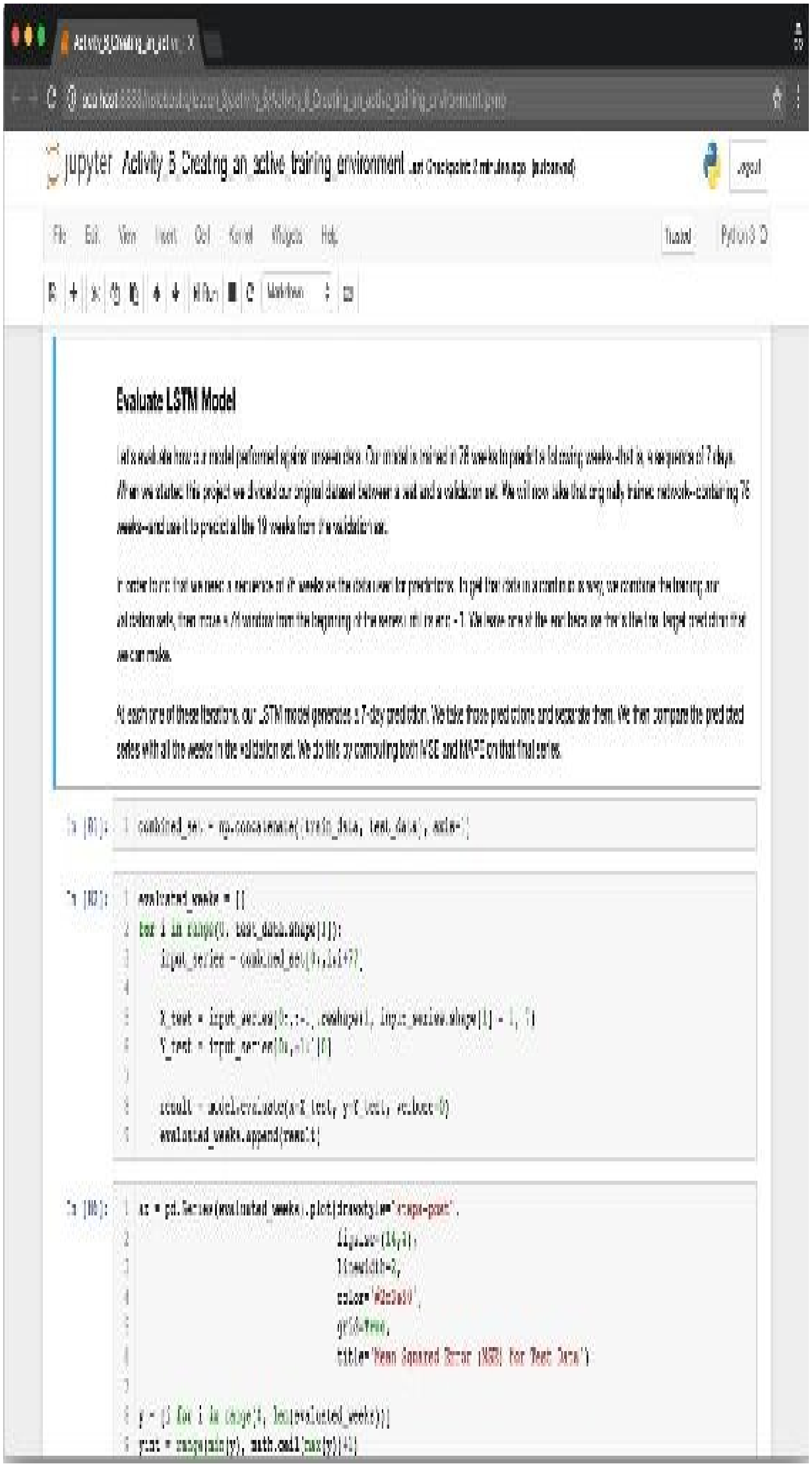
In this activity, we create a training environment for our neural network that facilitates both its training and evaluation. This environment is particularly important to our next lesson, in which we search for an optimal combination of hyperparameters.

First, we will start both a Jupyter Notebook instance and a TensorBoard instance. Both of these instances can remain open for the remainder of this activity.

1. Using your terminal, navigate to the directory `lesson_3/activity_6` and execute the following code to start a Jupyter Notebook instance:

   ```
   $ jupyter notebook
   ```

2. Open the URL provided by the application in your browser and open the Jupyter Notebook named `Activity_6_Creating_an_active_training_environment.ipynb`:


Activity_8_Creating_an_activ X
localhost:8888/notebooks/lesson_3/activity_8/Activity_8_Creating_an_active_training_environment.ipynb
jupyter Activity_8_Creating an active training environment Last Checkpoint: 2 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Markdown

Evaluate LSTM Model

Let's evaluate how our model performed against unseen data. Our model is trained in 76 weeks to predict the following week—that is, a sequence of 7 days. When we started this project we divided our original dataset between a test and a validation set. We will now take that originally trained network—containing 76 weeks—and use it to predict all the 19 weeks from the validation set.

In order to do that we need a sequence of 76 weeks as the data used for predictions. To get that data in a continuous way, we combine the training and validation sets, then move a 76 window from the beginning of the series until its end - 1. We leave one at the end because that's the final target prediction that we can make.

At each one of these iterations, our LSTM model generates a 7-day prediction. We take those predictions and separate them. We then compare the predicted series with all the weeks in the validation set. We do this by computing both MSE and MAPE on that final series.

```
In [80]: combined_set = np.concatenate((train_data, test_data), axis=1)
```

```
In [82]: evaluated_weeks = []
         for i in range(0, test_data.shape[1]):
             input_series = combined_set[0:,i:i+77]

             X_test = input_series[0:,:-1].reshape(1, input_series.shape[1] - 1, 7)
             Y_test = input_series[0:,-1:][0]

             result = model.evaluate(x=X_test, y=Y_test, verbose=0)
             evaluated_weeks.append(result)
```

```
In [86]: ax = pd.Series(evaluated_weeks).plot(drawstyle="steps-post",
                                              figsize=(14,4),
                                              linewidth=2,
                                              color='#2c3e50',
                                              grid=True,
                                              title='Mean Squared Error (MSE) for Test Data')

         y = [i for i in range(0, len(evaluated_weeks))]
         yint = range(min(y), math.ceil(max(y))+1)
```

Figure 7: Jupyter Notebook highlighting the section Evaluate LSTM Model

3. Also using your terminal, start a TensorBoard instance by executing the following command:

   ```
   $ cd ./lesson_3/activity_6/
   $ tensorboard --logdir=logs/
   ```

4. Open the URL that appears on the screen and leave that browser tab open, as well.
5. Now, load both the training (`train_dataset.csv`) and the test set (`test_dataset.csv`), and also our previously compiled model (`bitcoin_lstm_v0.h5`), into the Notebook.
6. Load the train and test datasets in the Jupyter Notebook instance using:

   ```
   $ train =
   pd.read_csv('data/train_dataset.csv')
   $ test =
   pd.read_csv('data/test_dataset.csv')
   ```

7. Also, load our previously compiled model using the following command:

   ```
   $ model =
   load_model('bitcoin_lstm_v0.h5')
   ```

   Let us now evaluate how our model performed against test data. Our model is trained using 76 weeks to predict a week into the future—that is, the following sequence of seven days. When we built our first model, we divided our original dataset between a training and a test set. We will now take a combined version of both datasets (let's call it combined set) and move a sliding window of 76 weeks. At each window, we execute Keras' `model.evaluate()` method to evaluate how the network performed on that specific week.
8. Execute the cells under the header **Evaluate LSTM Model**. The key concept of these cells it to call the `model.evaluate()` method for each of the weeks in the test set. This line is the most important:

   ```
   $ result = model.evaluate(x=X_test,
   y=Y_test, verbose=0)
   ```

9. Each evaluation result is now stored in the variable `evaluated_weeks`. That variable is a simple array containing the

sequence of MSE predictions for every week in the test set. Go ahead and also plot these results:

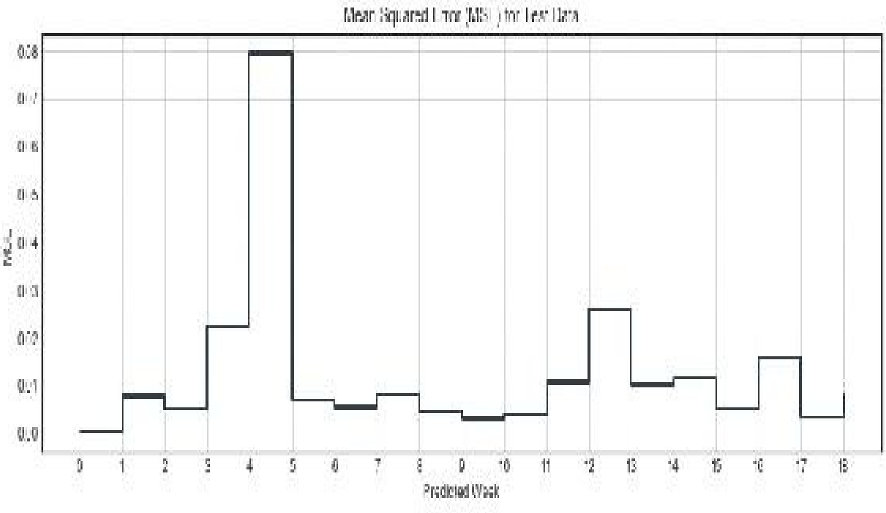


Figure 8: MSE results from the model.evaluate() method for each week of the test set

As discussed during our lesson, the MSE loss function is difficult to interpret. To facilitate our understanding of how our model is performing, we also call the method `model.predict()` on each week from the test set and compare its predicted results with the set's values.

10. Navigate to the section **Interpreting Model Results** and execute the code cells under the sub-header **Make Predictions**. Notice that we are calling the method `model.predict()`, but with a slightly different combination of parameters. Instead of using both `X` and `Y` values, we only use `X`:

```
predicted_weeks = []
 for i in range(0,
test_data.shape[1]):
 input_series =
combined_set[0:,i:i+76]

predicted_weeks.append(model.predict(i
nput_series))
```

At each window, we will issue predictions for the following week and store the results. We can now plot the normalized results alongside the normalized values from the test set, as shown in the following figure:

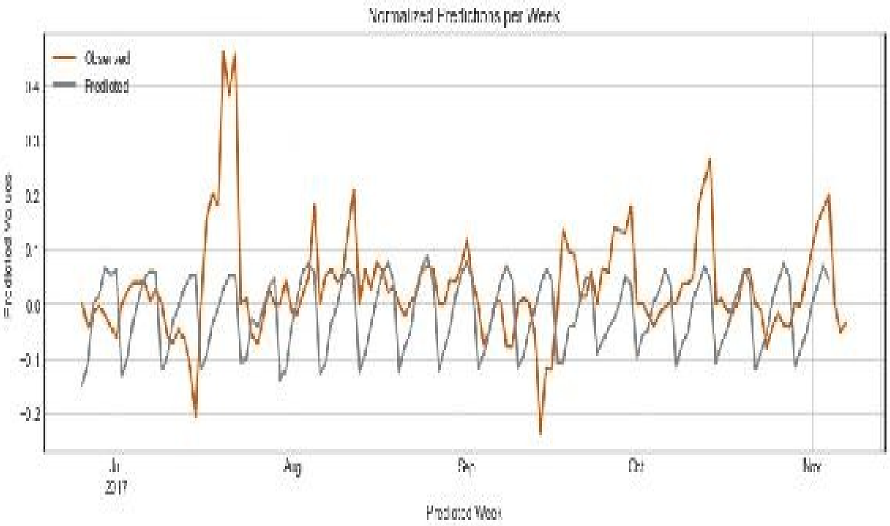


Figure 9: Plotting the normalized values returned from model.predict() for each week of the test set

We will also make the same comparisons but using denormalized values. In order to denormalize our data, we must first identify the equivalent week between the test set and the predictions. Then, we take the first price value for that week and use it to reverse the point-relative normalization equation from *Lesson 2, Model Architecture*.

11. Navigate to the header Denormalizing Predictions and execute all cells under that header.
12. In this section, we defined the function `denormalize(),` which performs the complete denormalization process. Different than other functions, this function takes in a Pandas DataFrame instead of a NumPy array. We do so for using dates as an index. This is the most relevant cell block from that header:

    ```
    predicted_close =
    predicted.groupby('iso_week').apply(
                                  lambda x:
    denormalize(observed, x))
    ```

Our denormalized results (as seen in the following figure) show that our model makes predictions that are close to the real Bitcoin prices. But how close?

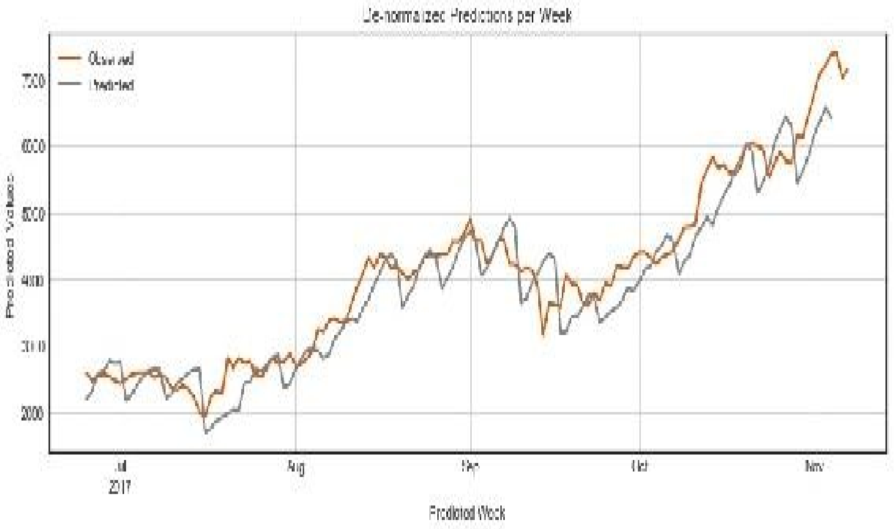


Figure 10: Plotting the denormalized values returned from model.predict() for each week of the test set

The LSTM network uses MSE values as its loss function. However, as discussed during the lesson, MSE values are difficult to interpret. To solve that, we implement two functions (loaded from the script `utilities.py`) that implement the functions RMSE and MAPE. Those functions add interpretability to our model by returning a measurement in the same scale that our original data used, and by comparing the difference in scale as a percentage.

13. Navigate to the header Denormalizing Predictions and load two functions from the `utilities.py` script:

```
from scripts.utilities import rmse,
mape
```

The functions from the script are actually really simple:

```
def mape(A, B):
    return np.mean(np.abs((A - B) /
A)) * 100

def rmse(A, B):
    return
np.sqrt(np.square(np.subtract(A,
B)).mean())
```

Each function is implemented using NumPy's vector-wise operations. They work well in vectors of the same length. They are designed to be applied on a complete set of results.

Using the `mape()` function, we can now understand that our model predictions are about 8.4 percent away from the prices from the test set. This is equivalent to a root mean squared error (calculated using the `rmse()` function) of about $399.6.

Before moving on to the next section, go back into the Notebook and find the header Re-train Model with TensorBoard. You may have noticed that we created a helper function called `train_model()`. This function is a wrapper around our model that trains (using `model.fit()`) our model, storing its respective results under a new directory. Those results are then used by TensorBoard as a discriminator, in order to display statistics for different models.

14. Go ahead and modify some of the values for the parameters passed to the `model.fit()` function (try epochs, for instance). Now, run the cells that load the model into memory from disk (this will replace your trained model):

    ```
    model =
    load_model('bitcoin_lstm_v0.h5')
    ```

15. Now, run the `train_model()` function again, but with different parameters, indicating a new run version:

    ```
    train_model(X=X_train, Y=Y_validate,
    version=0, run_number=0)
    ```

> **NOTE**
>
> For the reference solution, use the `Code/Lesson-3/activity_6` folder.

In this section, we learned how to evaluate a network using loss functions. We learned that loss functions are key elements of neural networks, as they evaluate the performance of a network at each epoch and are the starting point for the propagation of adjustments back into layers and nodes. We also explored why some loss functions can be difficult to interpret (for instance, the MSE) and developed a strategy using two other functions—RMSE and MAPE—to interpret the predicted results from our LSTM model.

Most importantly, this lesson concludes with an active training environment. We now have a system that can train a deep learning model and evaluate its results continuously. This will be key when we move to optimizing our network in the next session.

HYPERPARAMETER OPTIMIZATION

We have trained a neural network to predict the next seven days of Bitcoin prices using the preceding 76 weeks of prices. On average, that model issues predictions that are about 8.4 percent distant from real Bitcoin prices.

This section describes common strategies for improving the performance of neural network models:

- Adding or removing layers and changing the number of nodes
- Increasing or decreasing the number of training epochs
- Experimenting with different activation functions
- Using different regularization strategies

We will evaluate each modification using the same active learning environment developed by the end of the *Model Evaluation* section, measuring how each one of these strategies may help us develop a more precise model.

Layers and Nodes - Adding More Layers

Neural networks with single hidden layers can perform fairly well on many problems. Our first Bitcoin model (`bitcoin_lstm_v0`) is a good example: it can predict the next seven days of Bitcoin prices (from the test set) with error rates of about 8.4 percent using a single LSTM layer. However, not all problems can be modeled with single layers.

The more complex the function that you are working to predict is, the higher the likelihood that you will need to add more layers. A good intuition to determine whether adding new layers is a good idea is to understand what their role in a neural network is.

Each layer creates a model representation of its input data. Earlier layers in the chain create lower-level representations, and later layers, higher-level.

While that description may be difficult to translate into real-world problems, its practical intuition is simple: when working with complex functions that have different levels of representation, you may want to experiment with the addition of layers.

Adding More Nodes

The number of neurons that your layer requires is related to how both the input and output data are structured. For instance, if you are working to classify a 4 x 4 pixel image into one of two categories, one can start with a hidden layer that has 12 neurons (one for each available pixel) and an output layer that has only two (one for each predicted class).

It is common to add new neurons alongside the addition of new layers. Then, one can add a layer that has either the same number of neurons as the previous one, or a multiple of the number of neurons from the previous layer. For instance, if your first hidden layer has 12 neurons, you can experiment with adding a second layer that has either 12, 6, or 24.

Adding layers and neurons can have significant performance limitations. Feel free to experiment with adding layers and nodes. It is common to start with a smaller network (that is, a network with a small number of layers and neurons), then grow according to its performance gains.

If the above comes across as imprecise, your intuition is right. To quote Aurélien Géron, YouTube's former lead for video classification, *Finding the perfect amount of neurons is still somewhat of a black art*.

NOTE

Hands-on Machine Learning with Scikit-Learn and TensorFlow, by Aurelién Géron, published by O'Reilly, March 2017.

Finally, a word of caution: the more layers you add, the more hyperparameters you have to tune—and the longer your network will take to train. If your model is performing fairly well and not overfitting your data, experiment with the other strategies outlined in this lesson before adding new layers to your network.

Layers and Nodes - Implementation

We will now modify our original LSTM model by adding more layers. In LSTM models, one typically adds LSTM layers in a sequence, making a chain between LSTM layers. In our case, the new LSTM layer has the same number of neurons as the original layer, so we don't have to configure that parameter.

We will name the modified version of our model `bitcoin_lstm_v1`. It is good practice to name each one of the models in which one is attempting different hyperparameter configurations differently. This helps you to keep track of how each different architecture performs, and also to easily compare model differences in TensorBoard. We will compare all the different modified architectures at the end of this lesson.

NOTE

Before adding a new LSTM layer, we need to modify the parameter `return_sequences` to `True` on the first LSTM layer. We do this because the first layer expects a sequence of data with the same input as that of the first layer. When this parameter is set to `False,` the LSTM layer outputs the predicted parameters in a different, incompatible output.

Consider the following code example:

```
period_length = 7
number_of_periods = 76
batch_size = 1

  model = Sequential()
  model.add(LSTM(
      units=period_length,
      batch_input_shape=(batch_size,
number_of_periods, period_length),
      input_shape=(number_of_periods,
period_length),
      return_sequences=True,
stateful=False))

  model.add(LSTM(
      units=period_length,
      batch_input_shape=(batch_size,
number_of_periods, period_length),
      input_shape=(number_of_periods,
period_length),
      return_sequences=False,
stateful=False))

  model.add(Dense(units=period_length))
  model.add(Activation("linear"))

  model.compile(loss="mse",
optimizer="rmsprop")
```

Snippet 8: Adding a second LSTM layer to the original `bitcoin_lstm_v0 model`*, making it* `bitcoin_lstm_v1`

Epochs

Epochs are the number of times the network adjust its weights in response to data passing through and its loss function. Running a model for more epochs can allow it to learn more from data, but you also run the risk of overfitting.

When training a model, prefer to increase the epochs exponentially until the loss function starts to plateau. In the case of the `bitcoin_lstm_v0` model, its loss function plateaus at about 100 epochs.

Our LSTM model uses a small amount of data to train, so increasing the number of epochs does not affect its performance in significant ways. For instance, if one attempts to train it at 103 epochs, the model barely gains any improvements. This will not be the case if the model being trained uses enormous amounts of data. In those cases, a large number of epochs is crucial to achieve good performance.

I suggest you use the following association: the larger the date used to train your model, the more epochs it will need to achieve good performance.

Epochs - Implementation

Our Bitcoin dataset is rather small, so increasing the epochs that our model trains may have only a marginal effect on its performance. In order to have the model train for more epochs, one only has to change the `epochs` parameter in `model.fit()`:

```
number_of_epochs = 10**3
model.fit(x=X, y=Y, batch_size=1,
          epochs=number_of_epochs,
          verbose=0,
          callbacks=[tensorboard])
```

Snippet 9: Changing the number of epochs that our model trains for, making it `bitcoin_lstm_v2`

That change bumps our model to `v2`, effectively making it `bitcoin_lstm_v2`.

Activation Functions

Activation functions evaluate how much you need to *activate* individual neurons. They determine the value that each neuron will pass to the next element of the network, using both the input from the previous layer and the results from the loss function—or if a neuron should pass any values at all.

NOTE

Activation functions are a topic of great interest in the scientific community researching neural networks. For an overview of research currently being done on the topic and a more detailed review on how activation functions work, please refer to *Deep Learning* by Ian Goodfellow et. al., MIT Press, 2017.

TensorFlow and Keras provide many activation functions —and new ones are occasionally added. As an introduction, three are important to consider; let's explore each of them.

NOTE

This section has been greatly inspired by the article *Understanding Activation Functions in Neural Networks* by Avinash Sharma V, available at: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

Linear (Identity)

Linear functions only activate a neuron based on a constant value. They are defined by:

$$f(x) = c\ {}^{*}\ (0, x)$$

When *c = 1*, neurons will pass the values as-is, without modification by the activation function. The issue with using linear functions is that, due to the fact that neurons are activated linearly, chained layers now function as a single large layer. In other words, one loses the ability to construct networks with many layers, in which the output of one influences the other:

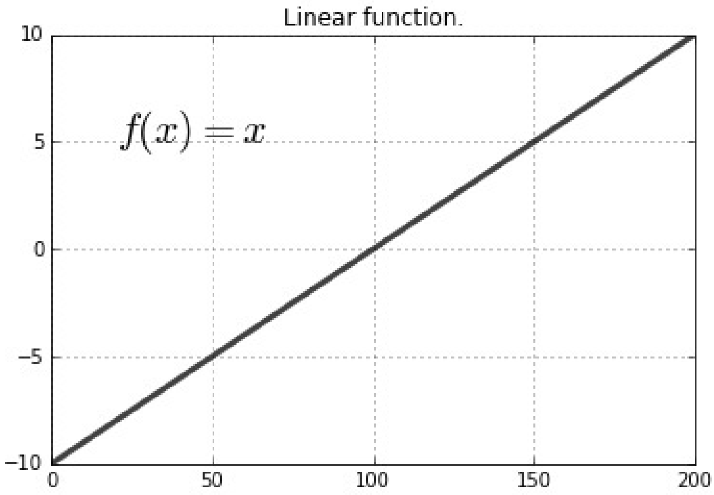


Figure 11: Illustration of a linear function

The use of linear functions is generally considered obsolete for most networks.

Hyperbolic Tangent (Tanh)

Tanh is a non-linear function, and is represented by the following formula:

$$f(x) = \frac{2}{2 + e^{-2x}} - 1$$

This means that the effect they have on nodes is evaluated continuously. Also, because of its non-linearity, one can use this function to change how one layer influences the next layer in the chain. When using non-linear functions, layers activate neurons in different ways, making it easier to learn different representations from data. However, they have a sigmoid-like pattern which penalizes extreme node values repeatedly, causing a problem called vanishing gradients. Vanishing gradients have negative effects on the ability of a network to learn:

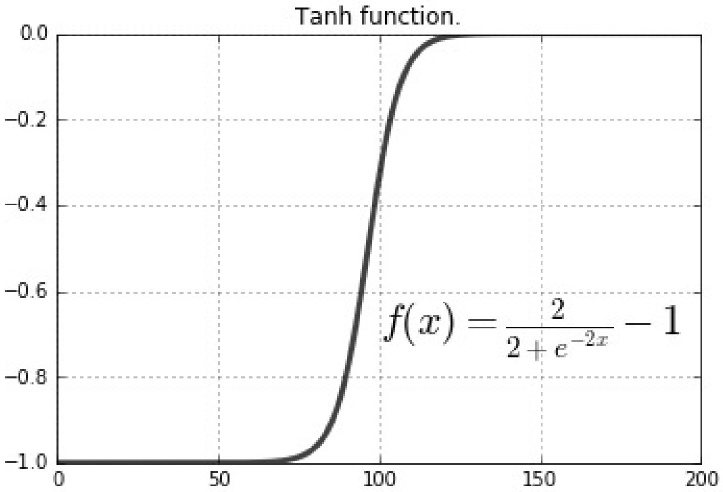


Figure 12: Illustration of a tanh function

Tanhs are popular choices, but due to fact that they are computationally expensive, ReLUs are often used instead.

Rectified Linear Unit

ReLUs have non-linear properties. They are defined by:

$$f(x) = max(0, x)$$

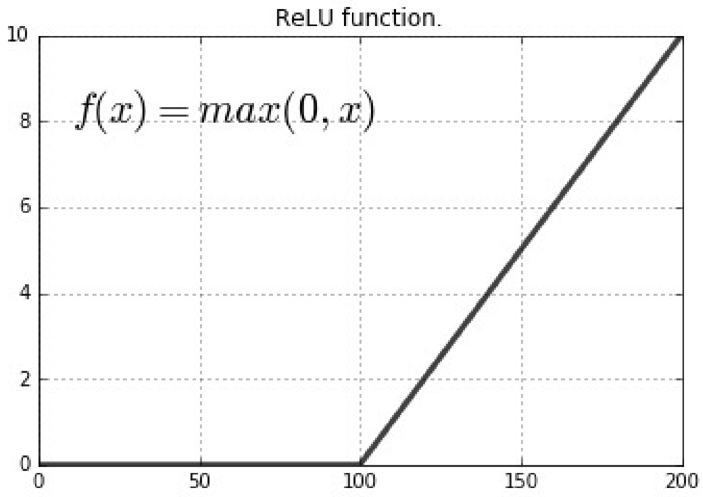


Figure 13: Illustration of a ReLU function

ReLU functions are often recommended as great starting points before trying other functions. ReLUs tend to penalize negative values. So, if the input data (for instance, normalized between `-1` and `1`) contains negative values, those will now be penalized by ReLUs. That may not be the intended behavior.

We will not be using ReLU functions in our network because our normalization process creates many negative values, yielding a much slower learning model.

Activation Functions - Implementation

The easiest way to implement activation functions in Keras is by instantiating the `Activation()` class and adding it to the `Sequential()` model. `Activation()` can be instantiated with any activation function available in Keras (for a complete list, see https://keras.io/activations/). In our case, we will use the `tanh` function. After implementing an activation function, we bump the version of our model to `v2`, making it `bitcoin_lstm_v3`:

```
model = Sequential()
```

```
model.add(LSTM(
      units=period_length,
      batch_input_shape=(batch_size,
number_of_periods, period_length),
      input_shape=(number_of_periods,
period_length),
      return_sequences=True,
stateful=False))

  model.add(LSTM(
      units=period_length,
      batch_input_shape=(batch_size,
number_of_periods, period_length),
      input_shape=(number_of_periods,
period_length),
      return_sequences=False,
stateful=False))

  model.add(Dense(units=period_length))
  model.add(Activation("tanh"))

  model.compile(loss="mse",
optimizer="rmsprop")
```

Snippet 10: Adding the activation function tanh to the `bitcoin_lstm_v2 model`*, making it* `bitcoin_lstm_v3`

There are a number of other activation functions worth experimenting with. Both TensorFlow and Keras provide a list of implemented functions in their respective official documentations. Before implementing your own, start with the ones already implemented in both TensorFlow and Keras.

Regularization Strategies

Neural networks are particularly prone to overfitting. Overfitting happens when a network learns the patterns of the training data but is unable to find generalizable patterns that can also be applied to the test data.

Regularization strategies refer to techniques that deal with the problem of overfitting by adjusting how the network learns. In this book, we discuss two common strategies: L2 and Dropout.

L2 Regularization

L2 regularization (or weight decay) is a common technique for dealing with overfitting models. In some models, certain parameters vary in great magnitudes. The L2 regularization penalizes such parameters, reducing the effect of these parameters on the network.

L2 regularizations use the λ parameter to determine how much to penalize a model neuron. One typically sets that to a very low value (that is, `0.0001`); otherwise, one risks eliminating the input from a given neuron completely.

Dropout

Dropout is a regularization technique based on a simple question: if one randomly takes away a proportion of nodes from layers, how will the other node adapt? It turns out that the remaining neurons adapt, learning to represent patterns that were previously handled by those neurons that are missing.

The dropout strategy is simple to implement and is typically very effective to avoid overfitting. This will be our preferred regularization.

Regularization Strategies – Implementation

In order to implement the dropout strategy using Keras, we import the `Dropout()` class and add it to our network immediately after each LSTM layer. This addition effectively makes our network `bitcoin_lstm_v4`:

```
model = Sequential()
  model.add(LSTM(
      units=period_length,
      batch_input_shape=(batch_size,
number_of_periods, period_length),
      input_shape=(number_of_periods,
period_length),
      return_sequences=True,
stateful=False))

  model.add(Dropout(0.2))

  model.add(LSTM(
      units=period_length,
      batch_input_shape=(batch_size,
number_of_periods, period_length),
      input_shape=(number_of_periods,
period_length),
      return_sequences=False
```

```
         return_sequences=False,
stateful=False))

  model.add(Dropout(0.2))

  model.add(Dense(units=period_length))
  model.add(Activation("tanh"))

  model.compile(loss="mse",
optimizer="rmsprop")
```

Snippet 11: In this snippet, we add the `Dropout()` *step to our model* `(bitcoin_lstm_v3)`*, making it* `bitcoin_lstm_v4`

One could have used the L2 regularization instead of Dropout. In order to do that, simply instantiate the `ActivityRegularization()` class with the `L2` parameter set to a low value (`0.0001,` for instance). Then, place it in the place where the `Dropout()` class is added to the network. Feel free to experiment by adding that to the network while keeping both `Dropout()` steps, or simply replace all the `Dropout()` instances with `ActivityRegularization()` instead.

Optimization Results

All in all, we have created four versions of our model. Three of these versions were created by the application of different optimization techniques outlined in this lesson.

After creating all these versions, we now have to evaluate which model performs best. In order to do that, we use the same metrics used in our first model: MSE, RMSE, and MAPE. MSE is used to compare the error rates of the model on each predicted week. RMSE and MAPE are computed to make the model results easier to interpret.

Model	**MSE (last epoch)**	**RMSE (whole series)**	**MAPE (whole series)**	**Training Time**
bitcoin_lstm_v0	-	399.6	8.4 percent	-
bitcoin_lstm_v1	7.15*10-6	419.3	8.8 percent	49.3 s
bitcoin_lstm_v2	3.55*10-6	425.4	9.0 percent	1min 13s
bitcoin_lstm_v3	2.8*10-4	423.9	8.8 percent	1min 19s
bitcoin_lstm_v4	4.8*10-7	442.4	9.4 percent	1min 20s

Table 2: Model results for all models

Interestingly, our first model (`bitcoin_lstm_v0`) performed the best in nearly all defined metrics. We will be using that model to build our web application and continuously predict Bitcoin prices.

Activity 7 – Optimizing a Deep Learning Model

In this activity, we implement different optimization strategies to the model created in *Lesson 2*, *Model Architecture* (`bitcoin_lstm_v0`). That model achieves a MAPE performance on the complete denormalization test set of about 8.4 percent. We will try to reduce that gap.

1. Using your terminal, start a TensorBoard instance by executing the following command:

   ```
   $ cd ./lesson_3/activity_7/
   $ tensorboard --logdir=logs/
   ```

2. Open the URL that appears on the screen and leave that browser tab open, as well. Also, start a Jupyter Notebook instance with:

   ```
   $ jupyter notebook
   ```

   Open the URL that appears in a different browser window.
3. Now, open the Jupyter Notebook called `Activity_7_Optimizing_a_deep_learning_model.ipynb` and navigate to the title of the Notebook and import all required libraries.

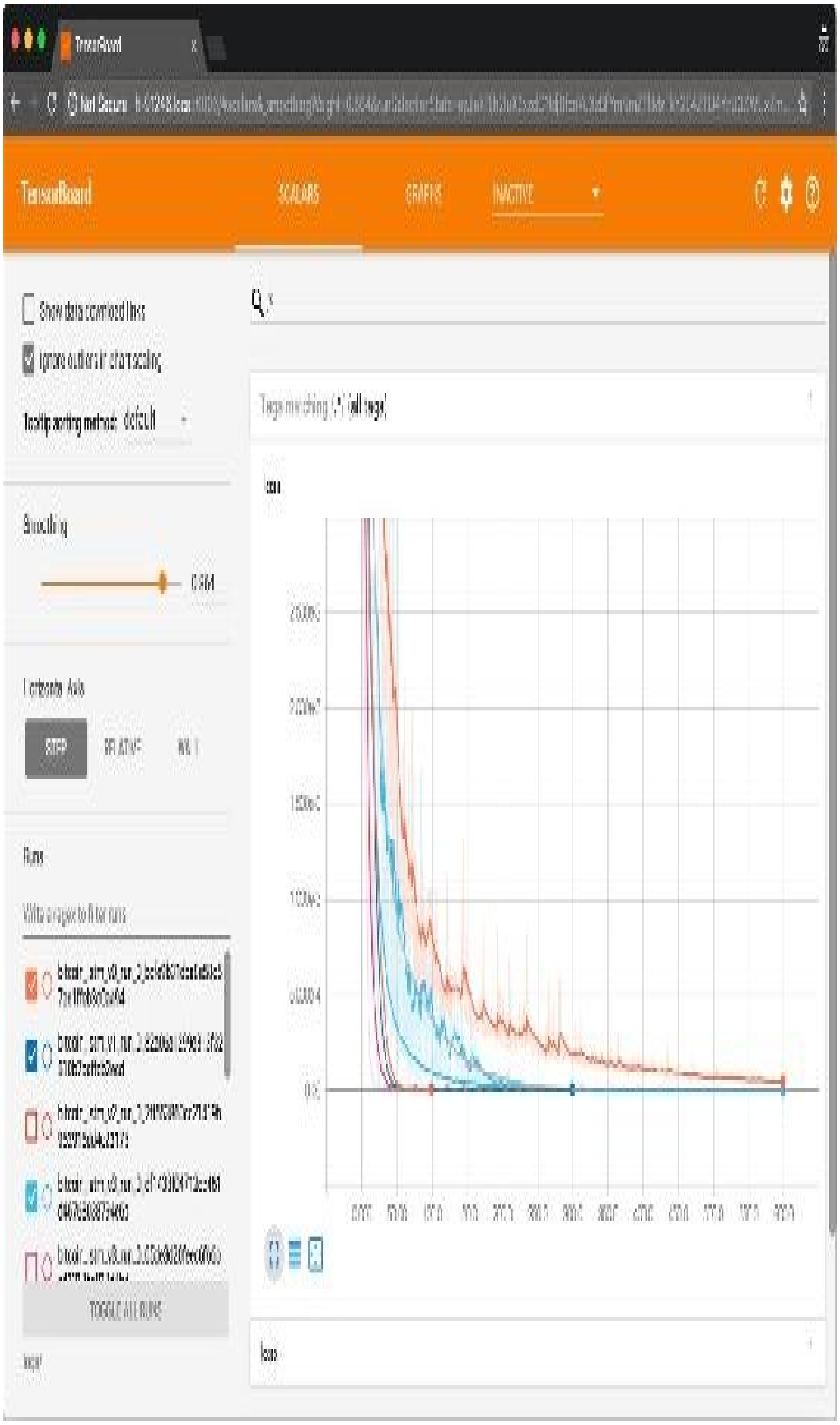
   We will load the train and test data like in previous activities. We will also split it into train and test groups using the utility function `split_lstm_input()`.

   In each section of this Notebook, we will implement new optimization techniques in our model. Each time we do so, we train a fresh model and store its trained instance in a variable that describes the model version. For instance, our first model, `bitcoin_lstm_v0,` is called `model_v0` in this Notebook. At the very end of the Notebook, we evaluate all models using MSE, RMSE, and MAPE.

4. Now, in the open Jupyter Notebook, navigate to the header **Adding Layers and Nodes**. You will recognize our first model in the next cell. This is the basic LSTM network that we built in *Lesson 2*, *Model Architecture*. Now, we have to add a new LSTM layer to this network.

   Using knowledge from this lesson, go ahead and add a new LSTM layer, compile, and train the model.

   While training your models, remember to frequently visit the running TensorBoard instance. You will be able to see each model run and compare the results of their loss functions there:



□ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

STEP RELATIVE WALL

Figure 14: Running the TensorBoard instance, which is displaying many different model runs. TensorBoard is really useful for tracking model training in real time.

5. Now, navigate to the header **Epochs**. In this section, we are

interested in exploring different magnitudes of epochs. Use the utility function `train_model()` to name different model versions and runs:

```
train_model(model=model_v0, X=X_train,
Y=Y_validate, epochs=100, version=0,
run_number=0)
```

Train the model with a few different epoch parameters.

At this point, you are interested in making sure the model doesn't overfit the training data. You want to avoid this, because if it does, it will not be able to predict patterns that are represented in the training data but have different representations in the test data.

After you are done experimenting with epochs, move to the next optimization technique: activation functions.

6. Now, navigate to the header **Activation Functions** in the Notebook. In this section, you only need to change the following variable:

   ```
   activation_function = "tanh"
   ```

   We have used the `tanh` function in this section, but feel free to try other activation functions. Review the list available at https://keras.io/activations/ and try other possibilities.

   Our final option is to try different regularization strategies. This is notably more complex and may take a few iterations to notice any gains—especially with so little data. Also, adding regularization strategies typically increases the training time of your network.
7. Now, navigate to the header **Regularization Strategies** in the Notebook. In this section, you need to implement the `Dropout()` regularization strategy. Find the right place to place that step and implement it in our model.
8. You can also try the L2 regularization here, as well (or combine both). Do the same as with `Dropout()`, but now using `ActivityRegularization(l2=0.0001)`.

   Finally, let's evaluate our models using RMSE and MAPE:
9. Now, navigate to the header **Evaluate Models** in the Notebook. In this section, we will evaluate the model predictions for the next 19 weeks of data in the test set. Then, we will compute the RMSE and MAPE of the predicted series versus the test series.

We have implemented the same evaluation techniques from Activity 6, all wrapped in utility functions. Simply run all the cells from this section until the end of the notebook to see the results.

NOTE

For the reference solution, use the `Code/Lesson-3/activity_7` folder.

Take this opportunity to tweak the values for the preceding optimization techniques and attempt to beat the performance of that model.

Summary

In this Lesson, we learned how to evaluate our model using the metrics mean squared error (MSE), squared mean squared error (RMSE), and mean averaged percentage error (MAPE). We computed the latter two metrics in a series of 19 week predictions made by our first neural network model. We then learned that it was performing well.

We also learned how to optimize a model. We looked at optimization techniques typically used to increase the performance of neural networks. Also, we implemented a number of these techniques and created a few more models to predict Bitcoin prices with different error rates.

In the next lesson, we will be turning our model into a web application that does two things: re-trains our model periodically with new data, and is able to make predictions using an HTTP API interface.

Chapter 4. Productization

This lesson focuses on how to *productize* a deep learning model. We use the word *productize* to define the creation of a software product from a deep learning model that can be used by other people and applications.

We are interested in models that use new data when it becomes available, continuously learning patterns from new data and, consequently, making better predictions. We study two strategies to deal with new data: one that re-trains an existing model, and another that creates a completely new model. Then, we implement the latter strategy in our Bitcoin prices prediction model so that it can continuously predict new Bitcoin prices.

This lesson also provides an exercise of how to deploy a model as a web application. By the end of this lesson, we will be able to deploy a working web-application (with a functioning HTTP API) and modify it to our heart's content.

We use a web application as an example of how to deploy deep learning models because of its simplicity and prevalence (after all, web application are quite common), but many other possibilities are available.

Lesson Objectives

In this lesson, you will:

- Handle new data
- Deploy a model as a web application

Handling New Data

Models can be trained once in a set of data and can then be used to make predictions. Such static models can be very useful, but it is often the case that we want our model to continuously learn from new data—and to continuously get better as it does so.

In this section, we will discuss two strategies on how to retrain a deep learning model and how to implement them in Python.

Separating Data and Model

When building a deep learning application, the two most important areas are data and model. From an architectural point of view, we suggest that these two areas be separate. We believe that is a good suggestion because each of these areas include functions inherently separated from each other. Data is often required to be collected, cleaned, organized, and normalized; and models need to be trained, evaluated, and able to make predictions. Both of these areas are dependent, but are better dealt with separately.

As a matter of following that suggestion, we will be using two classes to help us build our web application: `CoinMarketCap()` and `Model()`:

- `CoinMarketCap()`: This is a class designed for fetching Bitcoin prices from the following website: http://www.coinmarketcap.com. This is the same place where our original Bitcoin data comes from. This class makes it easy to retrieve that data on a regular schedule, returning a Pandas DataFrame with the parsed records and all available historical data. `CoinMarketCap()` is our data component.
- `Model()`: This class implements all the code we have written so far into a single class. That class provides facilities for interacting with our previously trained models, and also allows for the making of predictions using denormalized data—which is much easier to understand. The `Model()` class is our model component.

These two classes are used extensively throughout our example application and define the data and model components.

DATA COMPONENT

The `CoinMarketCap()` class creates methods for retrieving and parsing data. It contains one relevant method, `historic()`, which is detailed in the following code:

```
@classmethod
  def historic(cls, start='2013-04-28',
stop=None,
               ticker='bitcoin',
return_json=False):

    start = start.replace('-', '')
    if not stop:
        stop =
datetime.now().strftime('%Y%m%d')

    base_url =
'https://coinmarketcap.com/currencies'
    url = '{}historical-10.
data/?start={}&end={}'.format(ticker,
start,         stop)
    r = requests.get(url)
```

Snippet 1: `historic()` *method from the* `CoinMarketCap()` *class. This method collects data from the CoinMarketCap website, parses it, and returns a Pandas DataFrame.*

The `historic()` class returns a Pandas `DataFrame`, ready to be used by the `Model()` class.

When working in other models, consider creating a program component (for example, a Python class) that fulfills the same functions the `CoinMarketCap()` class does. That is, create a component that fetches data from wherever it is available, parses that data, and makes it available in a usable format for your modeling component.

The `CoinMarketCap()` class uses the parameter `ticker` to determine what cryptocurrency to collect. `CoinMarketCap` has many other cryptocurrencies available, including very popular ones like Ethereum (`ethereum`) and Bitcoin Cash (`bitcoin-cash`). Use the `ticker` parameter to change the cryptocurrency and train a different model than using the Bitcoin model created in this book.

Model Component

The `Model()` class is where we implement the application's model component. That class contains five methods that implement all the different modeling topics from this book. These are:

- `build()`: Builds an LSTM model using Keras. This function works as a simple wrapper for a manually created model.
- `train()`: Trains model using data that the class was instantiated with.
- `evaluate()`: Makes an evaluation of the model using a set of loss functions.
- `save()`: Saves the model as a file locally.
- `predict()`: Makes and returns predictions based on an input sequence of weeks-ordered observations.

We use these methods throughout this lesson to work, train, evaluate, and issue predictions with our model. The `Model()` class is an example of how to wrap essential Keras functions into a web application. The preceding methods are implemented almost exactly as in previous lessons, but with syntactic sugar added for enhancing their interfaces. For example, the method `train()` is implemented in the following code:

```
 def train(self, data=None, epochs=300,
verbose=0, batch_size=1):
    self.train_history = self.model.fit(
            x=self.X, y=self.Y,
            batch_size=batch_size,
epochs=epochs,
            verbose=verbose,
shuffle=False)

    self.last_trained =
datetime.now().strftime('%Y-%m-%d
%H:%M:%S')
    return self.train_history
```

Snippet 2: `train()` *method from the* `Model()` *class. This method trains a model available in* `self.model` *using data from* `self.X` *and* `self.Y`*.*

In the preceding snippet, you will be able to notice that the `train()` method resembles the solution to *Activities 6* and *7* from *Lesson 3*, *Model Evaluation and Optimization*. The general idea is that each of the processes from the Keras workflow (build or design, train, evaluate, and predict) can easily be turned into distinct parts of a program. In our case, we made them into methods that can be invoked from the `Model()` class. This organizes our program and provides a series of constraints (such as on the model architecture or certain API parameters) which help us deploy our model in a stable environment.

In the next sections, we explore common strategies for dealing with new data.

Dealing with New Data

The core idea of machine learning models—neural networks included—is that they can learn patterns from data. Imagine that a model was trained with a certain dataset and it is now issuing predictions. Now, imagine that new data is available. What strategies can we employ so that a model can take advantage of the newly available data to learn new patterns and improve its predictions?

In this section, we discuss two strategies: retraining an old model and training a new model.

ReTraining an Old Model

With this strategy, we retrain an existing model with new data. Using this strategy, one can continuously adjust the model parameters to adapt to new phenomena. However, data used in later training periods may be significantly different to other, earlier data. Such differences may cause significant changes to the model parameters, making it learn new patterns and forget old patterns. This phenomenon is generally referred to as *catastrophic forgetting*.

NOTE

Catastrophic forgetting is a common phenomenon affecting neural networks. Deep learning researchers have been trying to tackle this problem for many years. DeepMind, a Google-owned deep learning research group from the United Kingdom, has made notable advancements in finding a solution. The article *Overcoming Catastrophic Forgetting in Neural Networks,* by et. al. is a good reference of such work. The paper is available at: https://arxiv.org/pdf/1612.00796.pdf.

The same interface used for training (`model.fit()`) for the first time can be used for training with new data:

```
X_train_new, Y_train_new = load_new_data()

      model.fit(x=X_train_new,
y=Y_train_new,
       batch_size=1, epochs=100,
       verbose=0)
```

Snippet 3: Snippet that implements a TensorBoard callback in our LSTM model

In Keras, when models are trained, their weight information is kept—this is the model's state. When one uses the `model.save()` method, that state is also saved. And when one invokes the method `model.fit()`, the model is retrained with the new dataset, using the previous state as a starting point.

In typical Keras models, this technique can be used without further issues. However, when working with LSTM models, this technique has one key limitation: the shape of both train and validation data must be the same. For example, our LSTM model (`bitcoin_lstm_v0`) uses 76 weeks to predict one week into the future. If we attempt to retrain the network with 77 weeks in the coming week, the model raises an exception with information regarding the incorrect shape of data.

One way of dealing with this is to arrange data in the format expected by the model. In our case, we would need to configure our model to predict a future week using 40 weeks. Using this solution, we first train the model with the first 40 weeks of 2017, then continue to retrain it over the following weeks until we reach week 50. We use the `Model()` class to perform this operation in the following code:

```
M = Model(data=model_data[0*7:7*40 + 7],
             variable='close',
             predicted_period_size=7)

  M.build()
  M.train()

  for i in range(1, 10 + 1):
M.train(model_data[i*7:7*(40 + i) + 7])
```

Snippet 4: Snippet that implements a retraining technique

This technique tends to be fast to train, and also tends to work well with series that are large. The next technique is easier to implement and works well in smaller series.

Training a New Model

Another strategy is to create and train a new model every time new data is available. This approach tends to reduce catastrophic forgetting, but training time increases as data increases. Its implementation is quite simple.

Using the Bitcoin model as an example, let's now assume that we have old data for 49 weeks of 2017, and that after a week, new data is available. We represent this with the variables `old_data` and `new_data` in the following quotes:

```
old_data = model_data[0*7:7*48 + 7]
        new_data = model_data[0*7:7*49 +
7]
```

```
M = Model(data=old_data,
     variable='close',
     predicted_period_size=7)

M.build()
M.train()
M = Model(data=new_data,
    variable='close',
    predicted_period_size=7)

M.build()
M.train()
```

Snippet 5: Snippet that implements a strategy for training a new model when new data is available

This approach is very simple to implement and tends to work well for small datasets. This will be the preferred solution for our Bitcoin price-predictions application.

Activity 8 – Dealing with New Data

In this activity, we retrain our model every time new data is available.

First, we start by importing `cryptonic`. Cryptonic is a simple software application developed for this book that implements all the steps up to this section using Python classes and modules. Consider Cryptonic as a template of how you could develop similar applications.

`cryptonic` is provided as a Python module alongside this activity. First, we will start a Jupyter Notebook instance, and then we will load the `cryptonic` package.

1. Using your terminal, navigate to the directory `lesson_4/activity_8` and execute the following code to start a Jupyter Notebook instance:

    ```
    $ jupyter notebook
    ```

2. Open the URL provided by the application in your browser and open the Jupyter Notebook named `Activity_8_Re_training_a_model_dynamically.ipynb`.

    Now, we will load both classes from `cryptonic: Model()` and `CoinMarketCap()`. These classes facilitate the process of manipulating our model and also the process of getting data from the website CoinMarketCap (https://coinmarketcap.com/).

3. In the Jupyter Notebook instance, navigate to the header **Fetching Real-Time Data**. We will now be fetching updated historical data from CoinMarketCap. Simply call the method:

    ```
    $ historic_data =
    CoinMarketCap.historic()
    ```

    The variable `historic_data` is now populated with a Pandas DataFrame that contains data up to today or yesterday. This is great and makes it easier to retrain our model when more data is available.

    The data contains practically the same variables from our earlier dataset. However, much of the data comes from an earlier period. Recent Bitcoin prices have gained a lot of volatility compared to the prices of a few years ago. Before using this data in our model, let's make sure to filter it to dates after January 1, 2017.

4. Using the Pandas API, filter the data for only the dates available in 2017:

    ```
    $ model_data = # filter the dataset
    using pandas here
    ```

    You should be able to do this by using the date variable as the filtering index. Make sure the data is filtered before you continue.

    The class `Model()` compiles all the code we have written so far in all of our activities. We will use that class to build, train, and evaluate our model in this activity.

5. Using the `Model()` class, we now train a model using the preceding

filtered data:

```
M = Model(data=model_data,
          variable='close',
          predicted_period_size=7)

M.build()
M.train()
M.predict(denormalized=True)
```

The preceding steps showcase the complete workflow when using the `Model()` class for training a model.

> **NOTE**
>
> For the reference solution, use the `Code/Lesson-4/activity_8` folder.

Next, we'll focus on retraining our model every time more data is available. This re-adjusts the weights of the network to new data.

In order to do this, we have configured our model to predict a week using 40 weeks. We now want to use the remaining 10 full weeks to create overlapping periods of 40 weeks that include one of those 10 weeks at a time, and retrain the model for every one of those periods.

6. Navigate to the header **ReTrain Old Model** in the Jupyter Notebook. Now, complete the range function and the `model_data` filtering parameters, using an index to split the data in overlapping groups of seven days. Then, retrain our model and collect the results:

```
results = []
for i in range(A, B):
    M.train(model_data[C:D])
    results.append(M.evaluate())
```

The variables `A`, `B`, `C`, and `D` are placeholders. Use integers to create overlapping groups of seven days in which the overlap is of one day.

After you have retrained your model, go ahead and invoke the `M.predict(denormalized=True)` function and appreciate the results.

Next, we'll focus on creating and training a new model every time new data is available. In order to do this, we now assume that we have old data for 49 weeks of 2017, and after a week, we now have new data.

We represent this with the variables `old_data` and `new_data`.

7. Navigate to the header **Training a New Model** and split the data between the variables `old_data` and `new_data`:

   ```
   old_data = model_data[0*7:7*48 + 7]
   new_data = model_data[0*7:7*49 + 7]
   ```

8. Then, train the model with `old_data` first:

   ```
   M = Model(data=old_data,
             variable='close',
             predicted_period_size=7)
   M.build()
   M.train()
   ```

This strategy is about building the model from scratch and training it when new data is available. Go ahead and implement that in the following cells.

We now have all the pieces that we need in order to train our model dynamically. In the next section, we will deploy our model as a web application, making its predictions available in the browser via an HTTP API.

In this section, we learned about two strategies for training a model when new data is available:

- Retraining an old model
- Training a new model

The latter creates a new model that is trained with the full set of data, except the observations in the test set. The former trains a model once on available data, then continues to create overlapping batches to retrain that same model every time new data is available.

Deploying a Model as a Web Application

In this section, we will deploy our model as a web application. We will use an example web application—called "`cryptonic`"—to deploy our model, exploring its architecture so that we can make modifications in the future. The intention is to have you use this application as a starter for more complex applications; a starter that is fully working and can be expanded as you see fit.

Aside from familiarity with Python, this topic assumes familiarity with creating web applications. Specifically, we assume that you have some knowledge about web servers, routing, the HTTP protocol, and caching. You will be able to locally deploy the demonstrated cryptonic application without extensive knowledge of these topics, but learning these topics will make any future development much easier.

Finally, Docker is used to deploy our web applications, so basic knowledge of that technology is also useful.

Application Architecture and Technologies

In order to deploy our web applications, we will use the tools and technologies described on *Table 1*. Flask is key because it helps us create an HTTP interface to our model, allowing us to access an HTTP endpoint (such as `/predict`) and receive data back in a universal format. The other components are used because they are popular choices when developing web applications:

Tool or Technology	**Description**	**Role**
Docker	Docker is a technology used for working with applications packaged in the form of containers. Docker is an increasingly popular technology for building web applications.	Packages Python application and UI.
Flask	Flask is a micro-framework for building web applications in Python.	Creates application routes.
Vue.js	JavaScript framework that works by dynamically changing templates on the frontend based on data inputs from the backend.	Renders a user interface.
Nginx	Web server easily configurable to route traffic to Dockerized applications and handle SSL certificates for an HTTPS connection.	Routes traffic between user and Flask application.
Redis	Key-value database. It's a popular choice for implementing caching systems due to its simplicity and speed.	Cache API requests.

Table 1: Tools and technologies used for deploying a deep learning web application

These components fit together, as shown in the following figure:

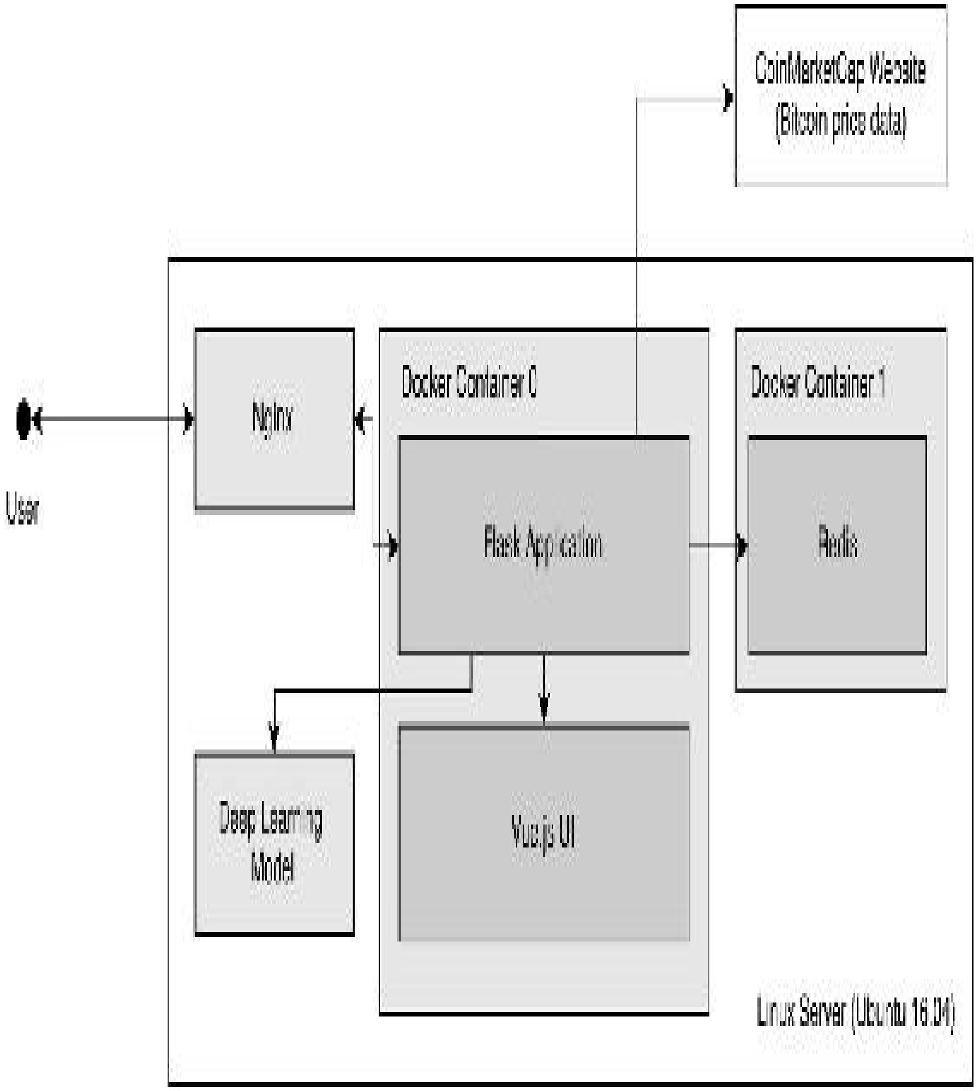


Figure 1: System architecture for the web application built in this project

A user visits the web application using their browser.

That traffic is then routed by Nginx to the Docker container containing the Flask application (by default, running on port `5000`). The Flask application has instantiated our Bitcoin model at startup. If a model has been given, it uses that model without training; if not, it creates a new model and trains it from scratch using data from CoinMarketCap.

After having a model ready, the application verifies if the request has been cached on Redis—if yes, it returns the cached data. If no cache exists, then it will go ahead and issue predictions which are rendered in the UI.

Deploying and Using Cryptonic

`cryptonic` is developed as a Dockerized application. In Docker terms, that means that the application can be built as a Docker image and then deployed as a Docker container in either a development or a production environment.

Docker uses files called `Dockerfile` for describing the rules for how to build an image and what happens when that image is deployed as a container. Cryptonic's Dockerfile is available in the following code:

```
FROM python:3.6
  COPY . /cryptonic
  WORKDIR "/cryptonic"
  RUN pip install -r requirements.txt
  EXPOSE 5000
  CMD ["python", "run.py"]
```

Snippet 7: Docker command for building a Docker image locally

A Dockerfile can be used to build a Docker image with the following command:

```
$ docker build --tag cryptonic:latest
```

This command will make the image `cryptonic:latest` available to be deployed as a container. The building process can be repeated on a production server, or the image can be directly deployed and then run as a container.

After an image has been built and is available, one can run the cryptonic application by using the command `docker run`, as shown in the following code:

```
$ docker run --publish 5000:5000 \
               --detach cryptonic:latest
```

Snippet 8: Example executing the docker run command in the terminal

The `--publish` flag binds port `5000` on `localhost` to port `5000` on the Docker container, and `--detach` runs the container as a daemon in the background.

In case you have trained a different model and would like to use that instead of training a new model, you can alter the `MODEL_NAME` environment variable on the `docker-`

`compose.yml`, as shown in Snippet 9. That variable should contain the filename of the model you have trained and want served (for example, `bitcoin_lstm_v1_trained.h5`)—it should also be a Keras model. If you do that, make sure to also mount a local directory into the `/models` folder. The directory that you decide to mount must have your model file.

The `cryptonic` application also includes a number of environment variables that you may find useful when deploying your own model:

- `MODEL_NAME`: Allows one to provide a trained model to be used by the application.
- `BITCOIN_START_DATE`: Determines which day to use as the starting day for the Bitcoin series. Bitcoin prices have a lot more variance in recent years than earlier ones. This parameter filters the data to only years of interest. The default is January 1, 2017.
- `PERIOD_SIZE`: Sets the period size in terms of days. The default is `7`.
- `EPOCHS`: Configures the number of epochs that the model trains on every run. The default is `300`.

These variables can be configured in the `docker-compose.yml` file, as shown in the following code:

```
version: "3"
 services:
 cache:
 image: cryptonic-cache:latest
 volumes: - $PWD/cache_data:/data
 networks:- cryptonic
 ports: - "6379:6379"

        environment:
            -
MODEL_NAME=bitcoin_lstm_v0_trained.h5
            - BITCOIN_START_DATE=2017-01-
01
            - EPOCH=300
            - PERIOD_SIZE=7
```

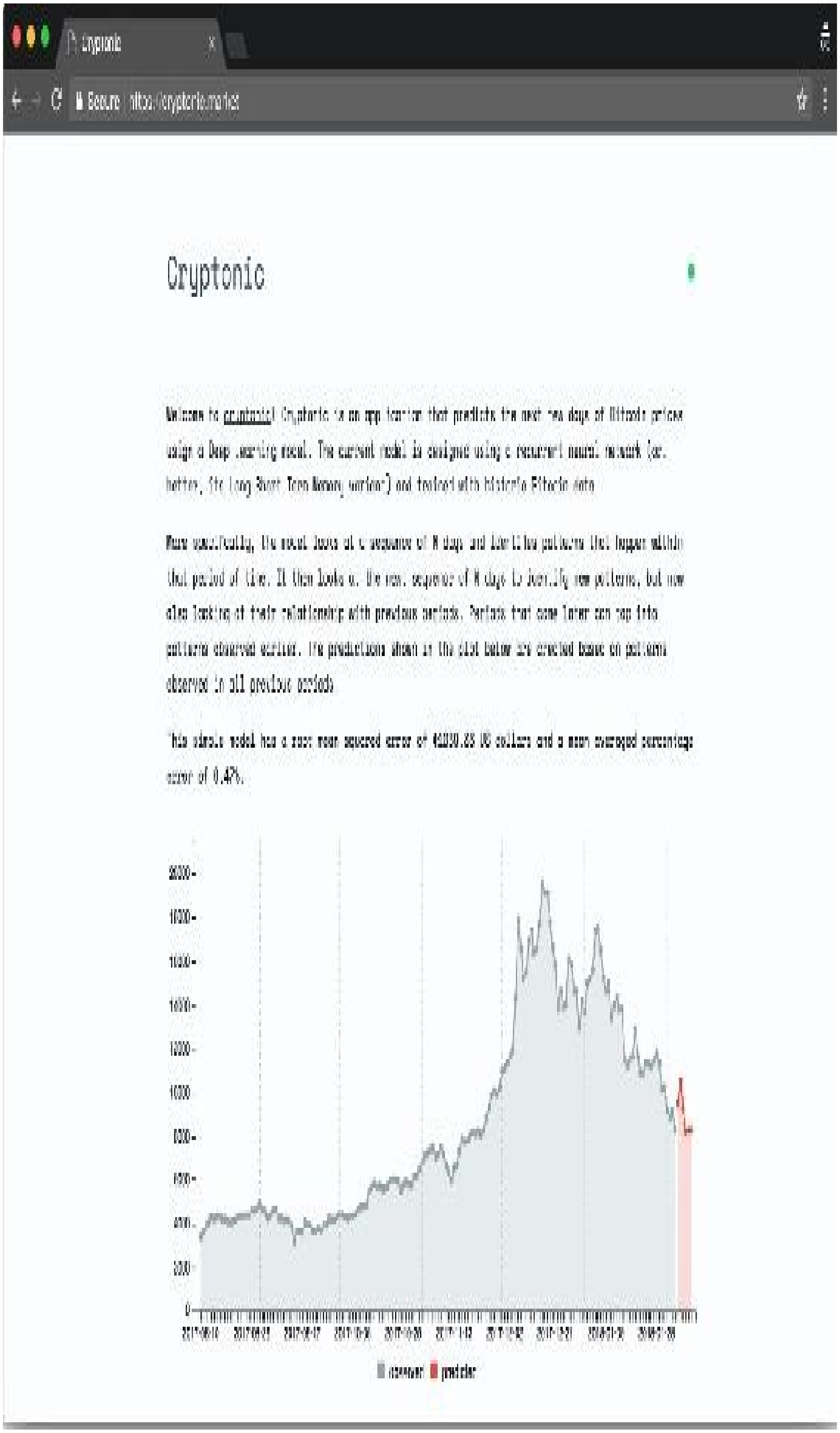
Snippet 9: docker-compose.yml file including environment variables

The easiest way to deploy `cryptonic` is to use the `docker-compose.yml` file from Snippet 9. This file contains all the specifications necessary for the application to run, including instructions on how to connect with the Redis cache, and what environment variables to use. After navigating to the location of the `docker-compose.yml` file, `cryptonic` can then be started with the command `docker-compose up`, as shown in the following code:

```
$ docker-compose up -d
```

Snippet 10: Starting a Docker application with docker-compose. The flag -d executes the application in the background.

After being deployed, `cryptonic` can be accessed on port `5000` via a web browser. The application has a simple user interface with a time-series plot depicting real historical prices (in other words, observed) and predicted future prices from the deep learning model (in other words, predicted). One can also read, in the text, both the RMSE and the MAPE calculated using the `Model().evaluate()` method:



Cryptonic

Welcome to cryptonic! Cryptonic is an application that predicts the next few days of Bitcoin prices using a Deep Learning model. The current model is designed using a recurrent neural network (or, better, its Long Short-Term Memory variant) and trained with historic Bitcoin data.

More specifically, the model looks at a sequence of N days and identifies patterns that happen within that period of time. It then looks at the next sequence of N days to identify new patterns, but now also looking at their relationship with previous periods. Periods that come later can map into patterns observed earlier. The predictions shown in the plot below are created based on patterns observed in all previous periods.

This simple model has a root mean squared error of 41030.23 US dollars and a mean averaged percentage error of 0.47%.

Figure 2: Screenshot of the deployed cryptonic application

Aside from its user interface (developed using Vue.js), the application has an HTTP API that makes predictions when invoked. The API has the endpoint `/predict`, which returns a JSON object containing the denormalized Bitcoin prices prediction for a week into the future:

```
{
    message: "API for making
predictions.",
    period_length: 7,
    result: [
      15847.7,
      15289.36,
      17879.07,
…
      17877.23,
      17773.08
    ],
    success: true,
    version: 1
  }
```

Snippet 11: Example JSON output from the /predict endpoint

The application can now be deployed in a remote server and used to continuously predict Bitcoin prices.

Activity 9 – Deploying a Deep Learning Application

In this activity, we deploy our model as a web application locally. This allows us to connect to the web application using a browser or to use another application through the application's HTTP API. Before we continue, make sure that you have the following applications installed and available in your computer:

- Docker (Community Edition) 17.12.0-ce or later
- Docker Compose (docker-compose) 1.18.0 or later

Both of the components above can be downloaded and installed in all major systems from the website: http://docker.com/. These are essential for completing this activity. Make sure these are available in your system before moving forward.

1. Using your terminal, navigate to the cryptonic directory and build the docker images for all the required components:

   ```
   $ docker build --tag cryptonic:latest .
   $ docker build --tag cryptonic-cache:latest ./ cryptonic-cache/
   ```

2. Those two commands build the two images that we will use in this application: cryptonic (containing the Flask application) and cryptonic-cache (containing the Redis cache).
3. After building the images, identify the `docker-compose.yml` file and open it in a text editor. Change the parameter `BITCOIN_START_DATE` to a date other than 2017-01-01:

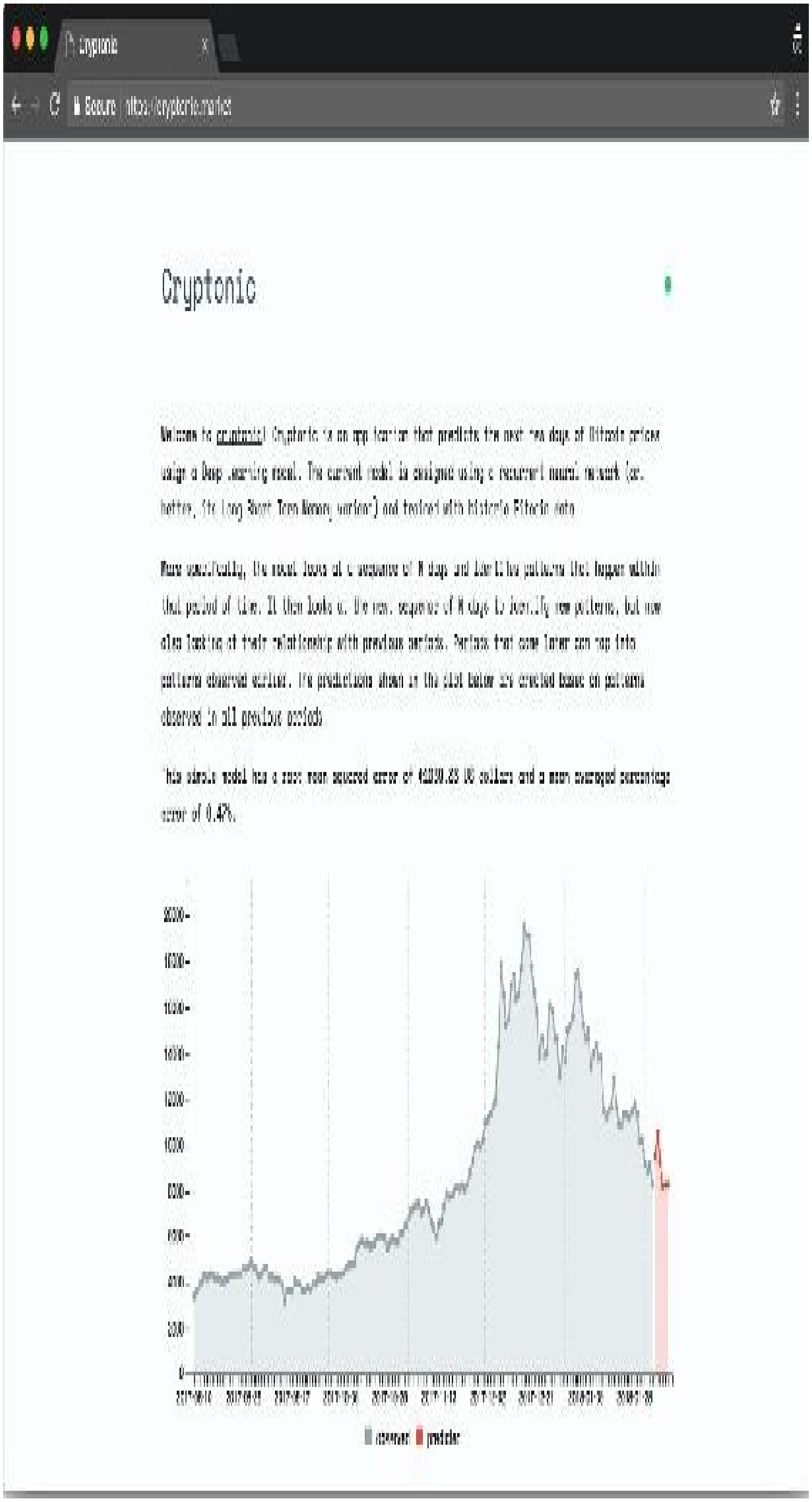
   ```
   BITCOIN_START_DATE = # Use other date here
   ```

4. As a final step, deploy your web application locally using `docker-compose,` as follows:

   ```
   docker-compose up
   ```

You should see a log of activity on your terminal, including training epochs from your model.

5. After the model has been trained, you can visit your application on `http://localhost:5000` and make predictions on `http://localhost:5000/predict`:



Cryptonic

Welcome to cryptonic! Cryptonic is an app toolkit that predicts the next few days of Bitcoin prices using a Deep Learning model. The current model is designed using a recurrent neural network (or, better, its Long Short Term Memory variant) and trained with historic Bitcoin data.

More specifically, the model looks at a sequence of N days and identifies patterns that happen within that period of time. It then looks at the next sequence of N days to identify new patterns, but now also looking at their relationship with previous periods. Periods that come later can map into patterns observed earlier. The predictions shown in the plot below are created based on patterns observed in all previous periods.

This simple model has a root mean squared error of 41030.23 US dollars and a mean averaged percentage error of 0.47%.

Figure 3: Screenshot of the cryptonic application deployed locally

NOTE

For the reference solution, use the `Code/Lesson-4/activity_9` folder.

Summary

This lesson concludes our journey into creating a deep learning model and deploying it as a web application. Our very last steps included deploying a model that predicts Bitcoin prices built using Keras and using a TensorFlow engine. We finished our work by packaging the application as a Docker container and deploying it so that others can consume the predictions of our model—as well as other applications via its API.

Aside from that work, you have also learned that there is much that can be improved. Our Bitcoin model is only an example of what a model can do (particularly LSTMs). The challenge now is two fold: how can you make that model perform better as time passes? And, what features can you add to your web application to make your model more accessible? Good luck and keep learning!

Index

A

- accuracy functions
  - using / Loss Functions, Accuracy, and Error Rates
- activation functions
  - about / Activation Functions
- active training environment
  - creating / Activity 6 – Creating an Active Training Environment
- AlphaGo algorithm
  - about / Successful Applications
- Artificial Neural Networks
  - about / What are Neural Networks?

B

- backpropagation / Function Approximation
  - about / Loss Functions, Accuracy, and Error Rates
- Bitcoin dataset
  - exploring / Activity 3 – Exploring the Bitcoin Dataset and Preparing Data for Model
  - preparing, for model / Activity 3 – Exploring the Bitcoin Dataset and Preparing Data for Model
- Bitcoin model
  - evaluating / Evaluating the Bitcoin Model

C

- CIFAR dataset
  - about / MNIST Dataset
  - reference / MNIST Dataset
- class Model()
  - about / Model Component
- CoinMarketCap() class / Separating Data and Model
- common architectures, deep learning
  - about / Common Architectures
  - convolutional neural networks / Convolutional Neural Networks
- contemporary neural network practice
  - about / Common Components and Operations of Neural Networks
- convolution / Convolutional Neural Networks
- convolutional layer
  - about / Training a Neural Network with TensorFlow
- Convolutional Neural Network / Training a Neural Network with TensorFlow
- convolutional neural networks (CNNs)
  - about / Common Architectures, Convolutional Neural Networks
- cryptonic
  - about / Deploying and Using Cryptonic
  - using / Deploying and Using Cryptonic
  - deploying / Deploying and Using Cryptonic
- cryptonic application

  - about / Deploying and Using Cryptonic
  - MODEL_NAME / Deploying and Using Cryptonic
  - BITCOIN_START_DATE / Deploying and Using Cryptonic
  - PERIOD_SIZE / Deploying and Using Cryptonic
  - EPOCHS / Deploying and Using Cryptonic
- custom neural network
  - training / Activity 2 – Exploring a Trained Neural Network

D

- data
  - and model, separating / Separating Data and Model
- data augmentation / Structuring Your Problem
- data component
  - about / Data Component
- data normalization
  - about / Data Normalization
- deep learning
  - about / What are Neural Networks?
  - limitations / Limitations of Deep Learning
  - environment, configuring / Configuring a Deep Learning Environment
  - model architecture, selecting / Choosing the Right Model Architecture
  - common architectures / Common Architectures
- deep learning application
  - deploying / Activity 9 – Deploying a Deep Learning Application
- deep learning model

    - optimizing / Activity 7 – Optimizing a Deep Learning Model
- deep learning system
    - about / From Data Preparation to Modeling
    - assembling / Activity 5 – Assembling a Deep Learning System
- DRL architecture
    - about / Deep Reinforcement Learning
- dropout strategy
    - about / Dropout
    - implementing / Regularization Strategies – Implementation

E

- epochs
    - about / Epochs
    - implementing / Epochs - Implementation
- error rates
    - using / Loss Functions, Accuracy, and Error Rates
- ethical considerations / Inherent Bias and Ethical Considerations
- examples, neural networks
    - self-driving vehicles / Successful Applications
    - image recognition / Successful Applications
- existing model
    - re-training / Re-Training an Old Model, Activity 8 – Dealing with New Data

F

- Fetching Real-Time Data / Activity 8 – Dealing with New Data

- forget gate
  - about / Recurrent Neural Networks
- fully connected layers
  - about / Training a Neural Network with TensorFlow

G

- generative adversarial networks (GANs)
  - about / Common Architectures, Generative Adversarial Networks
- Graphic Processing Units (GPUs) / Successful Applications

H

- hidden layers, neural networks
  - about / Common Components and Operations of Neural Networks
- Hyperbolic Tangent (Tanh)
  - about / Hyperbolic Tangent (Tanh)
- hyperparameter
  - about / Model Evaluation
- hyperparameter optimization
  - about / Lesson Objectives, Hyperparameter Optimization
  - layers, adding / Layers and Nodes - Adding More Layers
  - nodes, adding / Adding More Nodes
  - layers, implementing / Layers and Nodes - Implementation
  - epochs, adding / Epochs
  - epochs, implementing / Epochs - Implementation
  - activation functions / Activation Functions

    - linear functions / Linear (Identity)
    - Hyperbolic Tangent (Tanh) / Hyperbolic Tangent (Tanh)
    - Rectified Linear Unit / Rectified Linear Unit
    - regularization strategies / Regularization Strategies
    - optimization results / Optimization Results
    - optimization strategies, implementing / Activity 7 – Optimizing a Deep Learning Model

I

- inherent bias / Inherent Bias and Ethical Considerations

J

- Jupyter Notebooks
    - about / Jupyter Notebooks, Pandas, and NumPy

K

- Keras
    - about / Keras
    - using, as TensorFlow interface / Using Keras as a TensorFlow Interface
    - model components / Model Components

L

- L2 regularization
    - about / L2 Regularization
- layers categories, neural networks
    - input / Common Components and Operations of Neural Networks

  - hidden / Common Components and Operations of Neural Networks
  - output / Common Components and Operations of Neural Networks
- linear functions
  - about / Linear (Identity)
  - using / Linear (Identity)
- long-short term memory (LSTM) networks
  - about / Common Architectures
- loss functions
  - about / Loss Functions, Accuracy, and Error Rates
  - common loss functions / Loss Functions, Accuracy, and Error Rates
  - using / Different Loss Functions, Same Architecture
- LSTM model
  - predictions / Model Predictions
  - predictions, interpreting / Interpreting Predictions

M

- maximum normalization
  - about / Maximum and Minimum Normalization
- minimum normalization
  - about / Maximum and Minimum Normalization
- MNIST dataset
  - about / Exploring a Trained Neural Network, MNIST Dataset
  - reference / Exploring a Trained Neural Network
- model

  - deploying, as web application / Deploying a Model as a Web Application
- Model() class / Separating Data and Model
- model component
  - about / Model Component
  - build() / Model Component
  - train() / Model Component
  - evaluate() / Model Component
  - save() / Model Component
  - predict() / Model Component
- model evaluation
  - about / Lesson Objectives
- model evaluation metrics
  - implementing / Implementing Model Evaluation Metrics

N

- neural network graph
  - exploring / Activity 2 – Exploring a Trained Neural Network
- neural networks
  - about / What are Neural Networks?, Common Components and Operations of Neural Networks
  - applications / Successful Applications
  - examples / Successful Applications
  - working / Why Do Neural Networks Work So Well?
  - function approximation / Function Approximation
  - common components / Common Components and Operations of Neural Networks

  - operations / Common Components and Operations of Neural Networks
  - layers categories / Common Components and Operations of Neural Networks
  - training, with TensorFlow / Training a Neural Network with TensorFlow
  - evaluating, in MNIST dataset / Training a Neural Network with TensorFlow, Training a Neural Network, Testing Network Performance with Unseen Data
  - training / Training a Neural Network
  - time-series data, reshaping / Reshaping Time-Series Data
  - overfitting / Overfitting
- new data
  - handling / Handling New Data
  - dealing with / Dealing with New Data, Activity 8 – Dealing with New Data
- new model
  - training / Training a New Model, Activity 8 – Dealing with New Data
- Numpy
  - about / Jupyter Notebooks, Pandas, and NumPy

O

- optimization results
  - about / Optimization Results
- optimization strategies
  - implementing / Activity 7 – Optimizing a Deep Learning Model
- overfitting

  - about / Overfitting

P

- Pandas
  - about / Jupyter Notebooks, Pandas, and NumPy
- parameters
  - about / Model Evaluation
- point-relative normalization
  - about / Point-Relative Normalization
- predictions
  - making / Making Predictions
  - denormalizing / Making Predictions
- problem
  - structuring / Structuring Your Problem
- problem categories
  - classification / Problem Categories
  - regression / Problem Categories
- Python 3
  - about / Python 3

R

- Rectified Linear Unit (ReLUs)
  - about / Rectified Linear Unit
  - implementing / Activation Functions - Implementation
- recurrent neural networks (RNNs)

- about / Common Architectures, Recurrent Neural Networks
- regularization strategies
  - about / Regularization Strategies
  - L2 regularization / L2 Regularization
  - dropout / Dropout
- representation learning
  - about / Representation Learning

S

- software components, deep learning
  - Python 3 / Python 3
  - TensorFlow / TensorFlow
  - Keras / Keras
  - TensorBoard / TensorBoard
  - Jupyter Notebooks / Jupyter Notebooks, Pandas, and NumPy
  - Pandas / Jupyter Notebooks, Pandas, and NumPy
  - Numpy / Jupyter Notebooks, Pandas, and NumPy
  - verifying / Activity 1 – Verifying Software Components

T

- TensorBoard
  - about / TensorBoard, Using TensorBoard
  - using / Using TensorBoard
- TensorFlow
  - about / TensorFlow
- TensorFlow model

  - creating, with Keras / Activity 4 – Creating a TensorFlow Model Using Keras
- TensorFlow Playground application
  - about / Different Loss Functions, Same Architecture
- Tensor Processing Units (TPUs) / Successful Applications
- time-series data
  - reshaping / Reshaping Time-Series Data
- trained neural network
  - exploring / Exploring a Trained Neural Network, Activity 2 – Exploring a Trained Neural Network
- Transformer
  - about / Recurrent Neural Networks
- Transformer algorithm
  - about / Successful Applications

Z

- Z-score
  - about / Z-score