Long Short-Term Memory Networks With Python

Develop Sequence Prediction Models With Deep Learning

Jason Brownlee

MACHINE LEARNING MASTERY

Disclaimer

Copyright

Long Short-Term Memory Networks With Python

Edition: v1.6

Contents

Preface

This book was born out of one thought:

If I had to get a machine learning practitioner proficient with LSTMs in two weeks (e.g. capable of applying LSTMs to their own sequence prediction projects), what would I teach?

I had been researching and applying LSTMs for some time and wanted to write something on the topic, but struggled for months on how exactly to present it. The above question crystallized it for me and this whole book came together.

The above motivating question for this book is clarifying. It means that the lessons that I teach are focused only on the topics that you need to know in order to understand (1) what LSTMs are, (2) why we need LSTMs and (3) how to develop LSTM models in Python with Keras. It means that it is my job to give you the critical path.

- **From**: practitioner interested in LSTMs.
- **To**: practitioner that can confidently apply LSTMs to sequence prediction problems.

I want you to get proficient with LSTMs as quickly as you can. I want you using LSTMs on your project. This also means not covering some topics, even topics covered by "*everyone else*", like:

- **Theory**: The math of LSTMs is interesting even beautiful. It can deepen your understanding of what is going in within the LSTM fit and prediction processes. But it is not required in order to develop LSTM models and use them to make predictions and deliver value. A theoretical understanding of LSTMs is a nice-to-have next step. Not a prerequisite or first step to using LSTMs on your project.
- **Research**: There is a lot of interesting things going on in the field of LSTM and RNN research right now. Lots of interesting and fun ideas to talk about. But the coalface of research is noisy and it is not clear what techniques will actually survive and prove useful and what will be forgotten and never applied on real problems. This too is a nice-to have subsequent step and diving into the research can be something to consider after you know how to apply LSTMs to real problems.

The 14 lessons in this book are the fastest way that I know to get you proficient with LSTMs.

Jason Brownlee
2019

Part I

Introductions

Welcome

Welcome to *Long Short-Term Memory Networks With Python*. Long Short-Term Memory (LSTM) recurrent neural networks are one of the most interesting types of deep learning at the moment. They have been used to demonstrate world-class results in complex problem domains such as language translation, automatic image captioning, and text generation.

LSTMs are very different to other deep learning techniques, such as Multilayer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs), in that they are designed specifically for sequence prediction problems. I designed this book for you to rapidly discover what LSTMs are, how they work, and how you can bring this important technology to your own sequence prediction problems.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that know some applied machine learning and need to get good at LSTMs fast.

Maybe you want or need to start using LSTMs on your research project or on a project at work. This guide was written to help you do that quickly and efficiently by compressing years worth of knowledge and experience into a laser-focused course of 14 lessons. The lessons in this book assume a few things about you, such as:

- You know your way around basic Python.
- You know your way around basic NumPy.
- You know your way around basic scikit-learn.

For some bonus points, perhaps some of the below points apply to you. Don't panic if they don't.

- You may know how to work through a predictive modeling problem.
- You may know a little bit of deep learning.
- You may know a little bit of Keras.

This guide was written in the top-down and results-first machine learning style that you're used to from MachineLearningMastery.com.

About Your Outcomes

This book will teach you how to get results as a machine learning practitioner interested in using LSTMs on your project. After reading and working through this book, you will know:

1. What LSTMs are.
2. Why LSTMs are important.
3. How LSTMs work.
4. How to develop a suite of LSTM architectures.
5. How to get the most out of your LSTM models.

This book will NOT teach you how to be a research scientist and all the theory behind why LSTMs work. For that, I would recommend good research papers and textbooks. See the Further Reading section at the end of the first lesson for a good starting point.

How to Read This Book

This book was written to be read linearly from start to finish. That being said, if you know the basics and need help with a specific model type, then you can flip straight to that model and get started.

This book was designed for you to read on your workstation, on the screen, not on an eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding to your own deep learning projects. To get the most out of the book, I would recommend playing with the examples in each lesson. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed to be a 14-day crash course into LSTMs for machine learning practitioners. There are a lot of things you could learn about LSTMs, from theory to applications to Keras API. My goal is to take you straight to getting results with LSTMs in Keras with 14 laser-focused lessons.

I designed the lessons to focus on the LSTM models and their implementation in Keras. They give you the tools to both rapidly understand each model and apply them to your own sequence prediction problems. Each of the 14 lessons are designed to take you about one hour to read through and complete, excluding the extensions and further reading.

You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book was intended to be read and used, not to sit idle. I would recommend picking a schedule and sticking to it. The lessons are divided into three parts:

- **Part 1: Foundations**. The lessons in this section are designed to give you an understanding of how LSTMs work, how to prepare data, and the life-cycle of LSTM models in the Keras library.
- **Part 2: Models**. The lessons in this section are designed to teach you about the different types of LSTM architectures and how to implement them in Keras.
- **Part 3: Advanced**. The lessons in this section are designed to teach you how to get the most from your LSTM models.

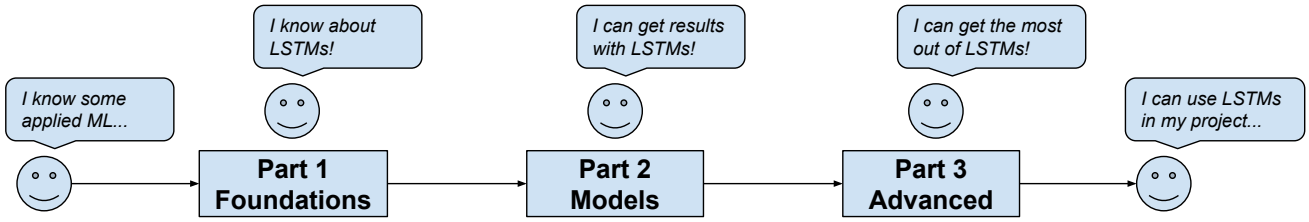

Figure 1: Overview of the 3-part book structure.

You can see that these parts provide a theme for the lessons with focus on the different types of LSTM models. Below is a breakdown of the 14 lessons organized by their part:

Part 1: Foundations

- **Lesson 1**: What are LSTMs.
- **Lesson 2**: How to Train LSTMs.
- **Lesson 3**: How to Prepare Data for LSTMs.
- **Lesson 4**: How to Develop LSTMs in Keras.
- **Lesson 5**: Models for Sequence Prediction.

Part 2: Models

- **Lesson 6**: How to Develop Vanilla LSTMs.
- **Lesson 7**: How to Develop Stacked LSTMs.
- **Lesson 8**: How to Develop CNN LSTMs.
- **Lesson 9**: How to Develop Encoder-Decoder LSTMs.
- **Lesson 10**: How to Develop Bidirectional LSTMs.
- **Lesson 11**: How to Develop Generative LSTMs.

Part 3: Advanced

- **Lesson 12**: How to Diagnose and Tune LSTMs.
- **Lesson 13**: How to Make Predictions with LSTMs.
- **Lesson 14**: How to Update LSTM Models.

You can see that each lesson has a targeted learning outcome. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and not get bogged down in the math or near-infinite number of configuration parameters.

These lessons were not designed to teach you everything there is to know about each of the LSTM models. They were designed to give you an understanding of how they work, how to use them on your projects the fastest way I know how: to learn by doing.

About Lessons

The core of this book are the lessons on the different LSTM models. The Foundation lessons build up to these lessons and the Advanced lessons complement the models once you know how to implement them. Each of the lessons in the Models part of the book follow a carefully designed structure, as follows:

- **Model Architecture**: Description of the model architecture, examples from seminal papers where it was developed or applied, and a presentation on how to generally implement it in Keras.
- **Problem Description**: Description of a test problem specifically designed to demonstrate the capability of the model architecture.
- **Define and Compile Model**: Description and example of how to define and compile the model architecture in Keras for the chosen problem.
- **Fit Model**: Description and example of how to fit the model in Keras on examples of the chosen problem.
- **Evaluate Model**: Description and example of how to evaluate the fit model in Keras on new examples of the chosen problem.
- **Predict with Model**: Description and example of how to make predictions with the fit model in Keras on new examples of the chosen problem.
- **Extensions**: Carefully chosen list of project ideas to build upon the lesson, ordered by increasing difficulty.
- **Further Reading**: Hand-picked list of research papers to deepen your understanding of the model architecture and its application and links to API documentation for the classes and functions highlighted in the example.

These sections were designed so that you understand the model architecture quickly, understand how to implement it with Keras with a worked example, and provide you with ways of deepening that understanding.

About LSTM Models

The LSTM network is the starting point. What you are really interested in is how to use the LSTM to address sequence prediction problems. The way that the LSTM network is used as layers in sophisticated network architectures. The way that you will get good at applying LSTMs is by knowing about the different useful LSTM networks and how to use them.

The whole middle section of this book focuses on teaching you about the different LSTM architectures. To give you an idea of what is coming, the list below summarizes each of the LSTM architectures presented in this book. Some have standard names and for some, I've assigned a standard name to help you differentiate them from other architectures.

- **Vanilla LSTM**. Memory cells of a single LSTM layer are used in a simple network structure.
- **Stacked LSTM**. LSTM layers are stacked one on top of another into deep networks.
- **CNN LSTM**. A convolutional neural network is used to learn features in spatial input like images and the LSTM can be used to support a sequence of images as input or generate a sequence in response to an image.
- **Encoder-Decoder LSTM**. One LSTM network encodes input sequences and a separate LSTM network decodes the encoding into an output sequence.
- **Bidirectional LSTM**. Input sequences are presented and learned both forward and backward.
- **Generative LSTM**. LSTMs learn the structure relationship in input sequences so well that they can generate new plausible sequences.

About Prediction Problems

The book uses small invented test problems to demonstrate each LSTM architecture instead of experimental or real-world datasets. This decision was very intentional and the reason behind this decision is as follows:

- **Size**. Demonstration problems must be small. I do not want you to have to download tens of gigabytes of text or images before being able to explore a new LSTM architecture. Small examples mean that we can get on with the example with modest time, memory and CPU requirements.
- **Complexity**. Demonstration problems must be easy to understand. I do not want you to get bogged down in the detail of a specific application example, especially if the application is image data and you are only interested in time series or text problems.
- **Tuning**. Demonstration problems must be able to scale in difficulty. It is important that the difficulty of the demonstration problems can be easily tuned, so that they provide a starting point for your own experimentation.

- **Focus**. The prediction problems are not the focus of this book. The goal of the book is to teach you how to use LSTMs, specifically the different useful LSTM architectures. The focus is not the application of LSTMs to one application area. The problem description sections of each lesson are already large enough.
- **Adaptability**. The examples must provide a template for your own projects. I want you be able to review each LSTM architecture demonstration and clearly see how it works. So much so, that I want you to be able to copy it into your own project, delete the functions for the test problem and start experimenting immediately on your own sequence prediction problems.

A total of 6 different sequence prediction problems were devised, one for each of the LSTM architectures demonstrated. Below is a summary of the sequence prediction problems used in the book. They are covered in much more detail later.

- **Echo Sequence Prediction Problem**. Given an input sequence of random integers, remember and predict the random integer at a specific input location. This is a sequence classification problem and is addressed with a many-to-one prediction model.
- **Damped Sine Wave Prediction Problem**. Given the input of multiple time steps of a damped sine wave, predict the next few time steps of the sequence. This is a sequence prediction problem or a multi-step time series forecasting problem and is addressed with a many-to-one prediction model (not a many-to-many model as you might expect).
- **Moving Square Video Prediction Problem**. Given a video sequence of images showing a square moving, predict the direction the square is moving. This is a sequence classification problem and is addressed with a many-to-one prediction model.
- **Addition Prediction Problem**. Given a sequence of characters representing the sum of multiple terms, predict the sequence of characters that represent the result of the mathematical operation. This is a sequence-to-sequence classification problem and is addressed with a many-to-many prediction model.
- **Cumulative Sum Prediction Problem**. Given an input sequence of random real values, predict a classification of whether the index has reached a cumulative sum threshold. This is a sequence-to-sequence classification problem or a time step classification problem and is addressed with a many-to-many prediction model.
- **Shape Generation Problem**. Given a catalog of example 2D shapes of a specific type, generate a new random shape that conforms to the general rules of the shape (e.g. consistent length and width of a rectangle). This is a sequence generation problem and is addressed with a one-to-one prediction model.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. For this reason, the examples are highly targeted.

- LSTM architectures are demonstrated on experimental problems to keep the focus on the model.
- Contrived demonstration problems can generally be scaled in terms of their complexity if you wish to make the examples more challenging.
- Model configurations used were discovered through trial and error are skillful, but not optimized. This leaves the door open for you to explore new and possibly better configurations.
- Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties required beyond the installation of the required packages.

A complete working example is presented with each chapter for you to inspect and copy-and-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead.

Neural network algorithms like LSTMs are stochastic. This means that they will make different predictions when the same model configuration is trained on the same training data. On top of that, each experimental problem in this book is based around generating stochastic input sequences. As a result, this means you will not get exactly the same sample output presented in this book. This is by design. I want you to get used to the stochastic nature of the neural network algorithms. If this bothers you, please note:

- You can re-run a given example a few times and your results should be close to the values reported.
- You can make the output consistent by fixing the NumPy random number seed.
- You can develop a robust estimate of the skill of a model by fitting and evaluating it multiple times and taking the average of the final skill score (highly recommended).

All code examples were tested with Python 2 and Python 3 with Keras 2. All code examples will run on modest and modern computer hardware and were executed on a CPU. No GPUs are required to run the presented examples, although a GPU would make the code run faster.

I am only human and there may be a bug in the sample code. If you discover a bug, please let me know so I can fix it and update the book.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.
- Books and book chapters.

- Webpages.
- API documentation.

Wherever possible, I try to list and link to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I try to list papers that are first to use a specific technique or first in a specific problem domain. These are not required reading, but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on http://ArXiv.org. You can search for and download any of the papers listed on Google Scholar Search https://scholar.google.com/. Wherever possible, I have tried to link to books on Amazon.

I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry, you are not alone.

- **Help with LSTMs?** If you need help with the technical aspects of LSTMs, see the *Further Reading* sections at the end of each lesson.
- **Help with Keras?** If you need help with using the Keras library, see the list of resources in Appendix A.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in Appendix B.
- **Help running large LSTM models?** I recommend renting time on Amazon Web Service (AWS) to run large models. If you need help getting started on AWS, see the tutorial in Appendix C.
- **Help in general?** You can shoot me an email. My details are in Appendix A.

Summary

Are you ready? Let's dive in!

Next up you will discover what LSTMs are and how they work.

Part II

Foundations

Chapter 1

What are LSTMs

1.0.1 Lesson Goal

The goal of this lesson is for you to develop a sufficiently high-level understanding of LSTMs so that you can explain what they are and how they work to a colleague or manager. After completing this lesson, you will know:

- What sequence predictions are and how they are different to general predictive modeling problems.
- The limitations of Multilayer Perceptrons for sequence prediction, the promise of Recurrent Neural Networks for sequence prediction, and how LSTMs deliver on that promise.
- Impressive applications of LSTMs to challenging sequence prediction problems and a caution about some of the limitations of LSTMs.

1.0.2 Lesson Overview

This lesson is divided into 6 parts; they are:

1. Sequence Prediction Problems.
2. Limitations of Multilayer Perceptrons.
3. Promise of Recurrent Neural Networks.
4. The Long Short-Term Memory Network.
5. Applications of LSTMs.
6. Limitations of LSTMs.

Let's get started.

1.1 Sequence Prediction Problems

Sequence prediction is different to other types of supervised learning problems. The sequence imposes an order on the observations that must be preserved when training models and making predictions. Generally, prediction problems that involve sequence data are referred to as sequence prediction problems, although there are a suite of problems that differ based on the input and output sequences. In this section we will take a look at the 4 different types of sequence prediction problems:

1. Sequence Prediction.
2. Sequence Classification.
3. Sequence Generation.
4. Sequence-to-Sequence Prediction.

But first, let's make sure we are clear on the difference between a set and a sequence.

1.1.1 Sequence

Often we deal with sets in applied machine learning such as a train or test set of samples. Each sample in the set can be thought of as an observation from the domain. In a set, the order of the observations is not important.

A sequence is different. The sequence imposes an explicit order on the observations. The order is important. It must be respected in the formulation of prediction problems that use the sequence data as input or output for the model.

1.1.2 Sequence Prediction

Sequence prediction involves predicting the next value for a given input sequence. For example:

```
Input Sequence: 1, 2, 3, 4, 5
Output Sequence: 6
```

Listing 1.1: Example of a sequence prediction problem.

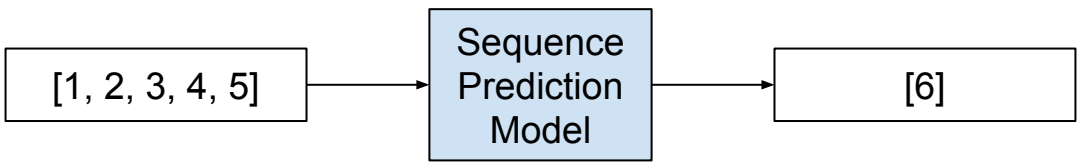

Figure 1.1: Depiction of a sequence prediction problem.

Sequence prediction may also generally be referred to as *sequence learning*. Technically, we could refer to all of the following problems as a type of sequence prediction problem. This can make things confusing for beginners.

> Learning of sequential data continues to be a fundamental task and a challenge in pattern recognition and machine learning. Applications involving sequential data may require prediction of new events, generation of new sequences, or decision making such as classification of sequences or sub-sequences.
>
> — *On Prediction Using Variable Order Markov Models*, 2004.

Generally throughout this book we will use "sequence prediction" to refer to the general class of prediction problems with sequence data. Nevertheless, in this section we will distinguish sequence prediction from other forms of prediction with sequence data as defining it as the prediction of the single next time step.

> Sequence prediction attempts to predict elements of a sequence on the basis of the preceding elements
>
> — *Sequence Learning: From Recognition and Prediction to Sequential Decision Making*, 2001.

Some examples of sequence prediction problems include:

- **Weather Forecasting**. Given a sequence of observations about the weather over time, predict the expected weather tomorrow.
- **Stock Market Prediction**. Given a sequence of movements of a security over time, predict the next movement of the security.
- **Product Recommendation**. Given a sequence of past purchases for a customer, predict the next purchase for a customer.

1.1.3 Sequence Classification

Sequence classification involves predicting a class label for a given input sequence. For example:

```
Input Sequence: 1, 2, 3, 4, 5
Output Sequence: "good"
```

Listing 1.2: Example of a sequence classification problem.

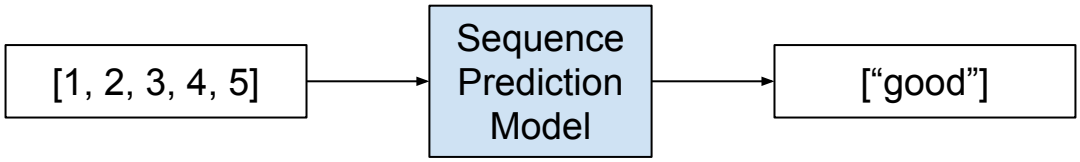

Figure 1.2: Depiction of a sequence classification problem.

> The objective of sequence classification is to build a classification model using a labeled dataset [...] so that the model can be used to predict the class label of an unseen sequence.

— *Discrete Sequence Classification, Data Classification: Algorithms and Applications*, 2015.

The input sequence may be comprised of real values or discrete values. In the latter case, such problems may be referred to as *discrete sequence classification* problems. Some examples of sequence classification problems include:

- **DNA Sequence Classification**. Given a DNA sequence of A, C, G, and T values, predict whether the sequence is for a coding or non-coding region.
- **Anomaly Detection**. Given a sequence of observations, predict whether the sequence is anomalous or not.
- **Sentiment Analysis**. Given a sequence of text such as a review or a tweet, predict whether the sentiment of the text is positive or negative.

1.1.4 Sequence Generation

Sequence generation involves generating a new output sequence that has the same general characteristics as other sequences in the corpus. For example:

```
Input Sequence: [1, 3, 5], [7, 9, 11]
Output Sequence: [3, 5 ,7]
```

Listing 1.3: Example of a sequence generation problem.

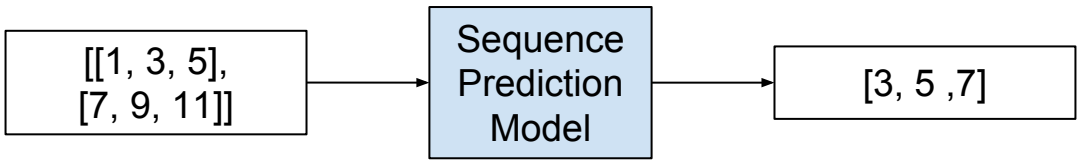

Figure 1.3: Depiction of a sequence generation problem.

> [recurrent neural networks] can be trained for sequence generation by processing real data sequences one step at a time and predicting what comes next. Assuming the predictions are probabilistic, novel sequences can be generated from a trained network by iteratively sampling from the network's output distribution, then feeding in the sample as input at the next step. In other words by making the network treat its inventions as if they were real, much like a person dreaming
>
> — *Generating Sequences With Recurrent Neural Networks*, 2013.

Some examples of sequence generation problems include:

- **Text Generation**. Given a corpus of text, such as the works of Shakespeare, generate new sentences or paragraphs of text that read they could have been drawn from the corpus.
- **Handwriting Prediction**. Given a corpus of handwriting examples, generate handwriting for new phrases that has the properties of handwriting in the corpus.

- **Music Generation**. Given a corpus of examples of music, generate new musical pieces that have the properties of the corpus.

Sequence generation may also refer to the generation of a sequence given a single observation as input. An example is the automatic textual description of images.

- **Image Caption Generation**. Given an image as input, generate a sequence of words that describe an image.

For example:

```
Input Sequence: [image pixels]
Output Sequence: ["man riding a bike"]
```

Listing 1.4: Example of a sequence generation problem.

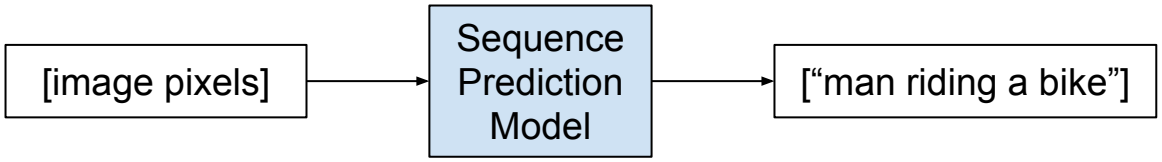

Figure 1.4: Depiction of a sequence generation problem for captioning an image.

> Being able to automatically describe the content of an image using properly formed English sentences is a very challenging task, but it could have great impact [...] Indeed, a description must capture not only the objects contained in an image, but it also must express how these objects relate to each other as well as their attributes and the activities they are involved in.
>
> — *Show and Tell: A Neural Image Caption Generator*, 2015.

1.1.5 Sequence-to-Sequence Prediction

Sequence-to-sequence prediction involves predicting an output sequence given an input sequence. For example:

```
Input Sequence: 1, 2, 3, 4, 5
Output Sequence: 6, 7, 8, 9, 10
```

Listing 1.5: Example of a sequence-to-sequence prediction problem.

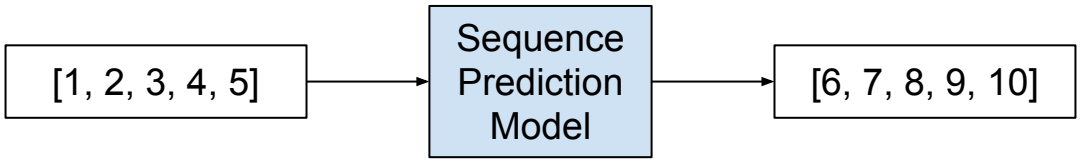

Figure 1.5: Depiction of a sequence-to-sequence prediction problem.

> Despite their flexibility and power, [deep neural networks] can only be applied to problems whose inputs and targets can be sensibly encoded with vectors of fixed dimensionality. It is a significant limitation, since many important problems are best expressed with sequences whose lengths are not known a-priori. For example, speech recognition and machine translation are sequential problems. Likewise, question answering can also be seen as mapping a sequence of words representing the question to a sequence of words representing the answer.
>
> — *Sequence to Sequence Learning with Neural Networks*, 2014.

Sequence-to-sequence prediction is a subtle but challenging extension of sequence prediction, where, rather than predicting a single next value in the sequence, a new sequence is predicted that may or may not have the same length or be of the same time as the input sequence. This type of problem has recently seen a lot of study in the area of automatic text translation (e.g. translating English to French) and may be referred to by the abbreviation *seq2seq*.

> seq2seq learning, at its core, uses recurrent neural networks to map variable-length input sequences to variable-length output sequences. While relatively new, the seq2seq approach has achieved state-of-the-art results in [...] machine translation.
>
> — *Multi-task Sequence to Sequence Learning*, 2016.

If the input and output sequences are a time series, then the problem may be referred to as *multi-step time series forecasting*. Some examples of sequence-to-sequence problems include:

- **Multi-Step Time Series Forecasting**. Given a time series of observations, predict a sequence of observations for a range of future time steps.
- **Text Summarization**. Given a document of text, predict a shorter sequence of text that describes the salient parts of the source document.
- **Program Execution**. Given the textual description program or mathematical equation predict the sequence of characters that describes the correct output.

1.2 Limitations of Multilayer Perceptrons

Classical neural networks called Multilayer Perceptrons, or MLPs for short, can be applied to sequence prediction problems. MLPs approximate a mapping function from input variables to output variables. This general capability is valuable for sequence prediction problems (notably time series forecasting) for a number of reasons.

- **Robust to Noise**. Neural networks are robust to noise in input data and in the mapping function and can even support learning and prediction in the presence of missing values.
- **Nonlinear**. Neural networks do not make strong assumptions about the mapping function and readily learn linear and nonlinear relationships.

More specifically, MLPs can be configured to support an arbitrarily defined but fixed number of inputs and outputs in the mapping function. This means that:

- **Multivariate Inputs**. An arbitrary number of input features can be specified, providing direct support for multivariate prediction.
- **Multi-Step Outputs**. An arbitrary number of output values can be specified, providing direct support for multi-step and even multivariate prediction.

This capability overcomes the limitations of using classical linear methods (think tools like ARIMA for time series forecasting). For these capabilities alone, feedforward neural networks are widely used for time series forecasting.

> ... one important contribution of neural networks - namely their elegant ability to approximate arbitrary nonlinear functions. This property is of high value in time series processing and promises more powerful applications, especially in the subfield of forecasting ...
>
> — *Neural Networks for Time Series Processing*, 1996.

The application of MLPs to sequence prediction requires that the input sequence be divided into smaller overlapping subsequences that are shown to the network in order to generate a prediction. The time steps of the input sequence become input features to the network. The subsequences are overlapping to simulate a window being slid along the sequence in order to generate the required output. This can work well on some problems, but it has 5 critical limitations.

- **Stateless**. MLPs learn a fixed function approximation. Any outputs that are conditional on the context of the input sequence must be generalized and frozen into the network weights.
- **Unaware of Temporal Structure**. Time steps are modeled as input features, meaning that network has no explicit handling or understanding of the temporal structure or order between observations.
- **Messy Scaling**. For problems that require modeling multiple parallel input sequences, the number of input features increases as a factor of the size of the sliding window without any explicit separation of time steps of series.
- **Fixed Sized Inputs**. The size of the sliding window is fixed and must be imposed on all inputs to the network.
- **Fixed Sized Outputs**. The size of the output is also fixed and any outputs that do not conform must be forced.

MLPs do offer great capability for sequence prediction but still suffer from this key limitation of having to specify the scope of temporal dependence between observations explicitly upfront in the design of the model.

> Sequences pose a challenge for [deep neural networks] because they require that the dimensionality of the inputs and outputs is known and fixed.
>
> — *Sequence to Sequence Learning with Neural Networks*, 2014

MLPs are a good starting point for modeling sequence prediction problems, but we now have better options.

1.3 Promise of Recurrent Neural Networks

The Long Short-Term Memory, or LSTM, network is a type of Recurrent Neural Network. Recurrent Neural Networks, or RNNs for short, are a special type of neural network designed for sequence problems. Given a standard feedforward MLP network, an RNN can be thought of as the addition of loops to the architecture. For example, in a given layer, each neuron may pass its signal laterally (sideways) in addition to forward to the next layer. The output of the network may feedback as an input to the network with the next input vector. And so on.

The recurrent connections add state or memory to the network and allow it to learn and harness the ordered nature of observations within input sequences.

> ... recurrent neural networks contain cycles that feed the network activations from a previous time step as inputs to the network to influence predictions at the current time step. These activations are stored in the internal states of the network which can in principle hold long-term temporal contextual information. This mechanism allows RNNs to exploit a dynamically changing contextual window over the input sequence history

— *Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling*, 2014

The addition of sequence is a new dimension to the function being approximated. Instead of mapping inputs to outputs alone, the network is capable of learning a mapping function for the inputs over time to an output. The internal memory can mean outputs are conditional on the recent context in the input sequence, not what has just been presented as input to the network. In a sense, this capability unlocks sequence prediction for neural networks.

> Long Short-Term Memory (LSTM) is able to solve many time series tasks unsolvable by feedforward networks using fixed size time windows.

— *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001

In addition to the general benefits of using neural networks for sequence prediction, RNNs can also learn and harness the temporal dependence from the data. That is, in the simplest case, the network is shown one observation at a time from a sequence and can learn what observations it has seen previously are relevant and how they are relevant to making a prediction.

> Because of this ability to learn long term correlations in a sequence, LSTM networks obviate the need for a pre-specified time window and are capable of accurately modelling complex multivariate sequences.

— *Long Short Term Memory Networks for Anomaly Detection in Time Series*, 2015

The promise of recurrent neural networks is that the temporal dependence and contextual information in the input data can be learned.

> A recurrent network whose inputs are not fixed but rather constitute an input sequence can be used to transform an input sequence into an output sequence while taking into account contextual information in a flexible way.

— *Learning Long-Term Dependencies with Gradient Descent is Difficult*, 1994.

There are a number of RNNs, but it is the LSTM that delivers on the promise of RNNs for sequence prediction. It is why there is so much buzz and application of LSTMs at the moment.

LSTMs have internal state, they are explicitly aware of the temporal structure in the inputs, are able to model multiple parallel input series separately, and can step through varied length input sequences to produce variable length output sequences, one observation at a time.

Next, let's take a closer look at the LSTM network.

1.4 The Long Short-Term Memory Network

The LSTM network is different to a classical MLP. Like an MLP, the network is comprised of layers of neurons. Input data is propagated through the network in order to make a prediction.

Like RNNs, the LSTMs have recurrent connections so that the state from previous activations of the neuron from the previous time step is used as context for formulating an output. But unlike other RNNs, the LSTM has a unique formulation that allows it to avoid the problems that prevent the training and scaling of other RNNs. This, and the impressive results that can be achieved, are the reason for the popularity of the technique.

The key technical historical challenge faced with RNNs is how to train them effectively. Experiments show how difficult this was where the weight update procedure resulted in weight changes that quickly became so small as to have no effect (vanishing gradients) or so large as to result in very large changes or even overflow (exploding gradients). LSTMs overcome this challenge by design.

> Unfortunately, the range of contextual information that standard RNNs can access is in practice quite limited. The problem is that the influence of a given input on the hidden layer, and therefore on the network output, either decays or blows up exponentially as it cycles around the network's recurrent connections. This shortcoming ... referred to in the literature as the vanishing gradient problem ... Long Short-Term Memory (LSTM) is an RNN architecture specifically designed to address the vanishing gradient problem.

— *A Novel Connectionist System for Unconstrained Handwriting Recognition*, 2009

The computational unit of the LSTM network is called the *memory cell*, *memory block*, or just *cell* for short. The term *neuron* as the computational unit is so ingrained when describing MLPs that it too is often used to refer to the LSTM memory cell. LSTM cells are comprised of weights and gates.

> The Long Short Term Memory architecture was motivated by an analysis of error flow in existing RNNs which found that long time lags were inaccessible to existing architectures, because backpropagated error either blows up or decays exponentially.
>
> An LSTM layer consists of a set of recurrently connected blocks, known as memory blocks. These blocks can be thought of as a differentiable version of the memory chips in a digital computer. Each one contains one or more recurrently connected memory cells and three multiplicative units - the input, output and forget gates - that provide continuous analogues of write, read and reset operations for the cells. ... The net can only interact with the cells via the gates.

— *Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures*, 2005.

1.4.1 LSTM Weights

A memory cell has weight parameters for the input, output, as well as an internal state that is built up through exposure to input time steps.

- **Input Weights**. Used to weight input for the current time step.
- **Output Weights**. Used to weight the output from the last time step.
- **Internal State**. Internal state used in the calculation of the output for this time step.

1.4.2 LSTM Gates

The key to the memory cell are the gates. These too are weighted functions that further govern the information flow in the cell. There are three gates:

- *Forget Gate*: Decides what information to discard from the cell.
- *Input Gate*: Decides which values from the input to update the memory state.
- *Output Gate*: Decides what to output based on input and the memory of the cell.

The forget gate and input gate are used in the updating of the internal state. The output gate is a final limiter on what the cell actually outputs. It is these gates and the consistent data flow called the *constant error carrousel* or CEC that keep each cell stable (neither exploding or vanishing).

> Each memory cell's internal architecture guarantees constant error flow within its constant error carrousel CEC... This represents the basis for bridging very long time lags. Two gate units learn to open and close access to error flow within each memory cell's CEC. The multiplicative input gate affords protection of the CEC from perturbation by irrelevant inputs. Likewise, the multiplicative output gate protects other units from perturbation by currently irrelevant memory contents.

— *Long Short-Term Memory*, 1997.

Unlike a traditional MLP neuron, it is hard to draw an LSTM memory unit cleanly. There are lines, weights, and gates all over the place. Take a look at some of the resources at the end of the chapter if you think pictures or equation-based description of LSTM internals would help further. We can summarize the 3 key benefits of LSTMs as:

- Overcomes the technical problems of training an RNN, namely vanishing and exploding gradients.
- Possesses memory to overcome the issues of long-term temporal dependency with input sequences.

- Process input sequences and output sequences time step by time step, allowing variable length inputs and outputs.

Next, let's take a look at some examples where LSTMs have address some challenging problems.

1.5 Applications of LSTMs

We are interested in LSTMs for the elegant solutions they can provide to challenging sequence prediction problems. This section provides 3 examples to give you a snapshot of the results that LSTMs are capable of achieving.

1.5.1 Automatic Image Caption Generation

Automatic image captioning is the task where, given an image, the system must generate a caption that describes the contents of the image. In 2014, there were an explosion of deep learning algorithms achieving very impressive results on this problem, leveraging the work from top models for object classification and object detection in photographs.

Once you can detect objects in photographs and generate labels for those objects, you can see that the next step is to turn those labels into a coherent sentence description. The systems involve the use of very large convolutional neural networks for the object detection in the photographs and then an LSTM to turn the labels into a coherent sentence.

Figure 1.6: Example of LSTM generated captions, taken from *Show and Tell: A Neural Image Caption Generator*, 2014.

1.5.2 Automatic Translation of Text

Automatic text translation is the task where you are given sentences of text in one language and must translate them into text in another language. For example, sentences of English as input and sentences of French as output. The model must learn the translation of words, the context where translation is modified, and support input and output sequences that may vary in length both generally and with regard to each other.

Type	Sentence
Our model	Ulrich UNK , membre du conseil d' administration du constructeur automobile Audi , affirme qu' il s' agit d' une pratique courante depuis des années pour que les téléphones portables puissent être collectés avant les réunions du conseil d' administration afin qu' ils ne soient pas utilisés comme appareils d' écoute à distance .
Truth	Ulrich Hackenberg , membre du conseil d' administration du constructeur automobile Audi , déclare que la collecte des téléphones portables avant les réunions du conseil , afin qu' ils ne puissent pas être utilisés comme appareils d' écoute à distance , est une pratique courante depuis des années .
Our model	" Les téléphones cellulaires , qui sont vraiment une question , non seulement parce qu' ils pourraient potentiellement causer des interférences avec les appareils de navigation , mais nous savons , selon la FCC , qu' ils pourraient interférer avec les tours de téléphone cellulaire lorsqu' ils sont dans l' air " , dit UNK .
Truth	" Les téléphones portables sont véritablement un problème , non seulement parce qu' ils pourraient éventuellement créer des interférences avec les instruments de navigation , mais parce que nous savons , d' après la FCC , qu' ils pourraient perturber les antennes-relais de téléphonie mobile s' ils sont utilisés à bord " , a déclaré Rosenker .
Our model	Avec la crémation , il y a un " sentiment de violence contre le corps d' un être cher " , qui sera " réduit à une pile de cendres " en très peu de temps au lieu d' un processus de décomposition " qui accompagnera les étapes du deuil " .
Truth	Il y a , avec la crémation , " une violence faite au corps aimé " , qui va être " réduit à un tas de cendres " en très peu de temps , et non après un processus de décomposition , qui " accompagnerait les phases du deuil " .

Figure 1.7: Example of English text translated to French comparing predicted to expected translations, taken from *Sequence to Sequence Learning with Neural Networks*, 2014.

1.5.3 Automatic Handwriting Generation

This is a task where, given a corpus of handwriting examples, new handwriting for a given word or phrase is generated. The handwriting is provided as a sequence of coordinates used by a pen when the handwriting samples were created. From this corpus, the relationship between the pen movement and the letters is learned and new examples can be generated. What is fascinating is that different styles can be learned and then mimicked. I would love to see this work combined with some forensic handwriting analysis expertise.

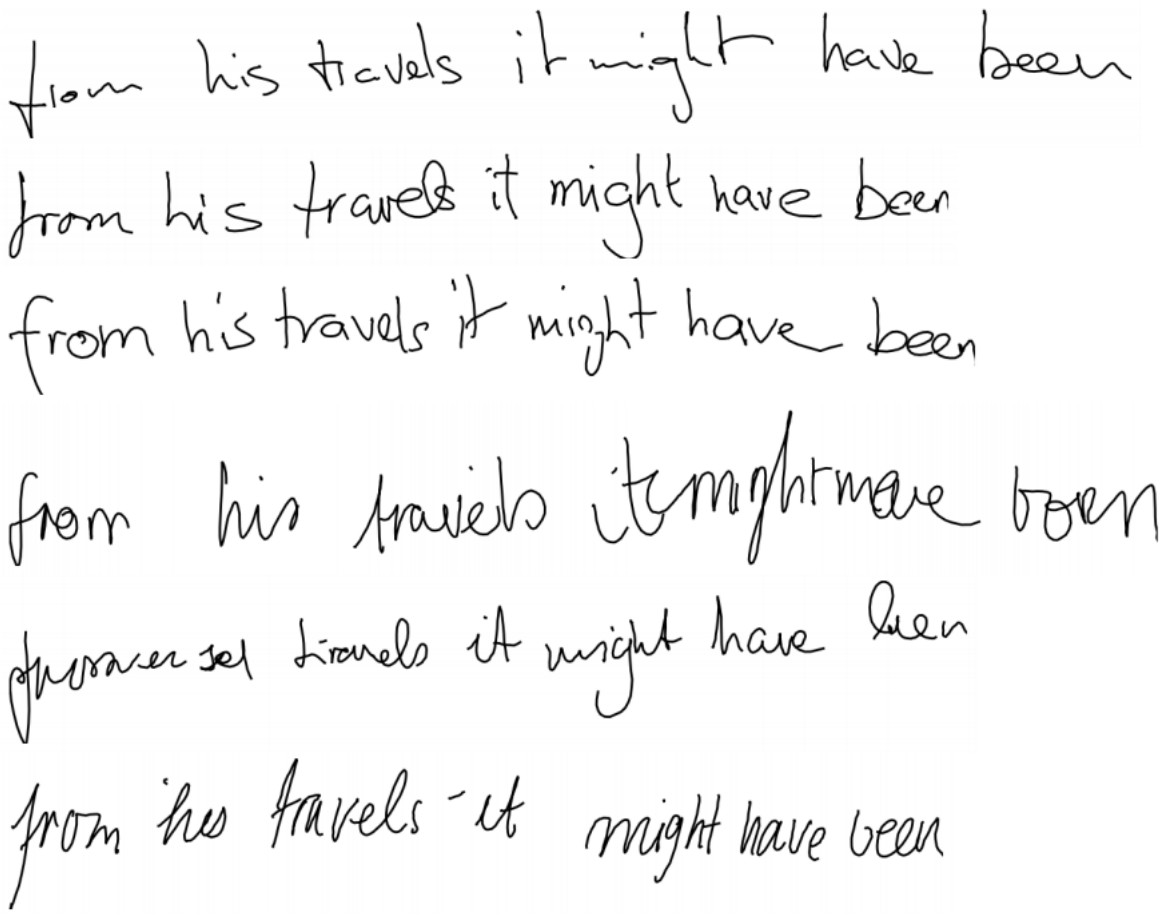

Figure 1.8: Example of LSTM generated captions, taken from *Generating Sequences With Recurrent Neural Networks*, 2014.

1.6 Limitations of LSTMs

LSTMs are very impressive. The design of the network overcomes the technical challenges of RNNs to deliver on the promise of sequence prediction with neural networks. The applications of LSTMs achieve impressive results on a range of complex sequence prediction problems. But LSTMs may not be ideal for all sequence prediction problems.

For example, in time series forecasting, often the information relevant for making a forecast is within a small window of past observations. Often an MLP with a window or a linear model may be a less complex and more suitable model.

> Time series benchmark problems found in the literature ... are often conceptually simpler than many tasks already solved by LSTM. They often do not require RNNs at all, because all relevant information about the next event is conveyed by a few recent events contained within a small time window.
>
> — *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001

An important limitation of LSTMs is the memory. Or more accurately, how memory can be abused. It is possible to force an LSTM model to remember a single observation over a very long number of input time steps. This is a poor use of LSTMs and requiring an LSTM model to remember multiple observations will fail.

This can be seen when applying LSTMs to time series forecasting where the problem is formulated as an autoregression that requires the output to be a function of multiple distant time steps in the input sequence. An LSTM may be forced to perform on this problem, but will generally be less efficient than a carefully designed autoregression model or reframing of the problem.

> Assuming that any dynamic model needs all inputs from t-tau ..., we note that the [autoregression]-RNN has to store all inputs from t-tau to t and overwrite them at the adequate time. This requires the implementation of a circular buffer, a structure quite difficult for an RNN to simulate.
>
> — *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001

The caution is that LSTMs are not a silver bullet and to carefully consider the framing of your problem. Think of the internal state of LSTMs as a handy internal variable to capture and provide context for making predictions. If your problem looks like a traditional autoregression type problem with the most relevant lag observations within a small window, then perhaps develop a baseline of performance with an MLP and sliding window before considering an LSTM.

> A time window based MLP outperformed the LSTM pure-[autoregression] approach on certain time series prediction benchmarks solvable by looking at a few recent inputs only. Thus LSTM's special strength, namely, to learn to remember single events for very long, unknown time periods, was not necessary
>
> — *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001

1.7 Further Reading

Below are a few must read papers on LSTMs if you are looking to dive deeper into the technical details of the algorithm.

1.7.1 Sequence Prediction Problems

- *Sequence on Wikipedia*.
  https://en.wikipedia.org/wiki/Sequence
- *On Prediction Using Variable Order Markov Models*, 2004.
- *Sequence Learning: From Recognition and Prediction to Sequential Decision Making*, 2001.
- Chapter 14, Discrete Sequence Classification, *Data Classification: Algorithms and Applications*, 2015.
  http://amzn.to/2tkM723

- *Generating Sequences With Recurrent Neural Networks*, 2013.
  https://arxiv.org/abs/1308.0850
- *Show and Tell: A Neural Image Caption Generator*, 2015.
  https://arxiv.org/abs/1411.4555
- *Multi-task Sequence to Sequence Learning*, 2016.
  https://arxiv.org/abs/1511.06114
- *Sequence to Sequence Learning with Neural Networks*, 2014.
  https://arxiv.org/abs/1409.3215
- *Recursive and direct multi-step forecasting: the best of both worlds*, 2012.

1.7.2 MLPs for Sequence Prediction

- *Neural Networks for Time Series Processing*, 1996.
- *Sequence to Sequence Learning with Neural Networks*, 2014.
  https://arxiv.org/abs/1409.3215

1.7.3 Promise of RNNs

- *Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling*, 2014.
- *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001.
- *Long Short Term Memory Networks for Anomaly Detection in Time Series*, 2015.
- *Learning Long-Term Dependencies with Gradient Descent is Difficult*, 1994.
- *On the difficulty of training Recurrent Neural Networks*, 2013.
  https://arxiv.org/abs/1211.5063

1.7.4 LSTMs

- *Long Short-Term Memory*, 1997.
- *Learning to forget: Continual prediction with LSTM*, 2000.
- *A Novel Connectionist System for Unconstrained Handwriting Recognition*, 2009.
- *Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures*, 2005.

1.7.5 LSTM Applications

- *Show and Tell: A Neural Image Caption Generator*, 2014.
  https://arxiv.org/abs/1411.4555
- *Sequence to Sequence Learning with Neural Networks*, 2014.
  https://arxiv.org/abs/1409.3215
- *Generating Sequences With Recurrent Neural Networks*, 2014.
  https://arxiv.org/abs/1308.0850

1.8 Extensions

Are you looking to deepen your understanding of LSTMs? This section lists some challenging extensions that you may wish to consider.

- Write a one-paragraph description of LSTMs for a novice practitioner.
- List 10 examples of sequence prediction problems that you can think of off the top of your head.
- Research and draw (or re-draw) a picture of an LSTM memory cell with all connection, weights, and gates.
- Research and list 10 more examples of interesting applications of LSTMs and clearly describe the sequence prediction problem, including inputs and outputs.
- Research and implement the equations for making a prediction with a single LSTM memory cell in a spreadsheet or in Python.

Post your extensions online and share the link with me, I'd love to see what you come up with!

1.9 Summary

In this lesson, you discovered the Long Short-Term Memory recurrent neural network for sequence prediction. Specifically, you learned:

- What sequence predictions are and how they are different to general predictive modeling problems.
- The limitations of Multilayer Perceptrons for sequence prediction, the promise of Recurrent neural networks for sequence prediction, and how LSTMs deliver on that promise.
- Impressive applications of LSTMs to challenging sequence prediction problems and a caution about some of the limitations of LSTMs.

Next, you will discover how LSTMs are trained using the Backpropagation Through Time training algorithm.

Chapter 2

How to Train LSTMs

2.0.1 Lesson Goal

The goal of this lesson is for you to understand the Backpropagation Through Time algorithm used to train LSTMs. After completing this lesson, you will know:

- What Backpropagation Through Time is and how it relates to the Backpropagation training algorithm used by Multilayer Perceptron networks.
- The motivations that lead to the need for Truncated Backpropagation Through Time, the most widely used variant in deep learning for training LSTMs.
- A notation for thinking about how to configure Truncated Backpropagation Through Time and the canonical configurations used in research and by deep learning libraries.

2.0.2 Lesson Overview

This lesson is divided into 6 parts; they are:

1. Backpropagation Training Algorithm.
2. Unrolling Recurrent Neural Networks.
3. Backpropagation Through Time.
4. Truncated Backpropagation Through Time.
5. Configurations for Truncated BPTT.
6. Keras Implementation of TBPTT.

Let's get started.

2.1 Backpropagation Training Algorithm

Backpropagation refers to two things:

- The mathematical method used to calculate derivatives and an application of the derivative chain rule.
- The training algorithm for updating network weights to minimize error.

It is this latter understanding of backpropagation that we are using in this lesson. The goal of the backpropagation training algorithm is to modify the weights of a neural network in order to minimize the error of the network outputs compared to some expected output in response to corresponding inputs. It is a supervised learning algorithm that allows the network to be corrected with regard to the specific errors made. The general algorithm is as follows:

1. Present a training input pattern and propagate it through the network to get an output.
2. Compare the predicted outputs to the expected outputs and calculate the error.
3. Calculate the derivatives of the error with respect to the network weights.
4. Adjust the weights to minimize the error.
5. Repeat.

2.2 Unrolling Recurrent Neural Networks

A simple conception of recurrent neural networks is as a type of neural network that takes inputs from previous time steps. We can demonstrate this with a diagram.

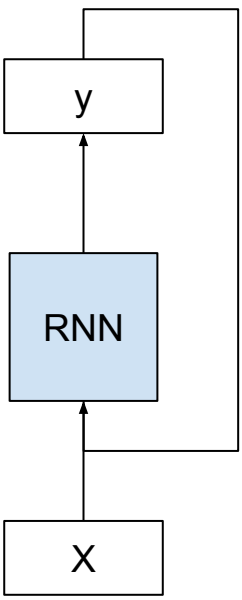

Figure 2.1: Example of a simple recurrent neural network.

RNNs are fit and make predictions over many time steps. As the number of time steps increases, the simple diagram with a recurrent connection begins to lose all meaning. We can simplify the model by unfolding or unrolling the RNN graph over the input sequence.

> A useful way to visualise RNNs is to consider the update graph formed by 'unfolding' the network along the input sequence.
>
> — *Supervised Sequence Labelling with Recurrent Neural Networks*, 2008.

2.2.1 Unfolding the Forward Pass

Consider the case where we have multiple time steps of input (`X(t), X(t+1), ...`), multiple time steps of intern state (`u(t), u(t+1), ...`), and multiple time steps of outputs (`y(t), y(t+1), ...`). We can unfold the network schematic into a graph without any cycles, as follows.

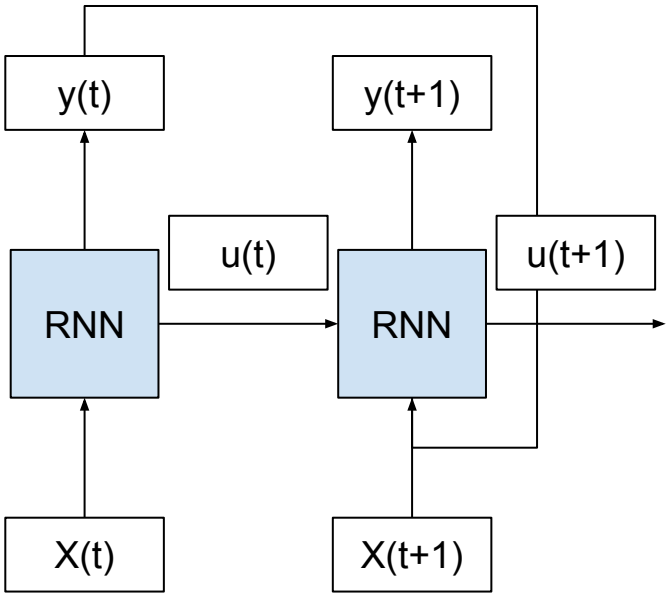

Figure 2.2: Example of an unfolded recurrent neural network.

We can see that the cycle is removed and that the output (`y(t)`) and internal state (`u(t)`) from the previous time step are passed on to the network as inputs for processing the next time step. Key in this conceptualization is that the network (`RNN`) does not change between the unfolded time steps. Specifically, the same weights are used for each time step and it is only the outputs and the internal states that differ. In this way, it is as though the whole network (topology and weights) are copied for each time step in the input sequence.

We can push this conceptualization one step further where each copy of the network may be thought of as an additional layer of the same feedforward neural network. The deeper layers take as input the output of the prior layer as well as a new input time step. The layers are in fact all copies of the same set of weights and the internal state is updated from layer to layer, which may be a stretch of this oft-used analogy.

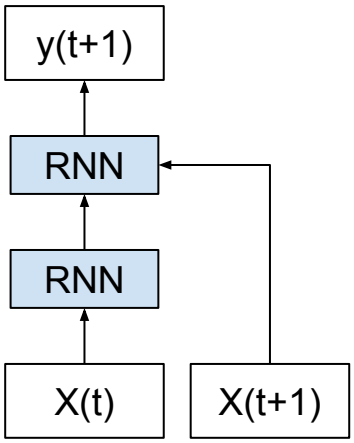

Figure 2.3: Example of an unfolded recurrent neural network with layers.

> RNNs, once unfolded in time, can be seen as very deep feedforward networks in which all the layers share the same weights.
>
> — *Deep learning*, Nature, 2015.

This is a useful conceptual tool and visualization to help in understanding what is going on in the network during the forward pass. It may or may not also be the way that the network is implemented by the deep learning library.

2.2.2 Unfolding the Backward Pass

The idea of network unfolding plays a bigger part in the way recurrent neural networks are implemented for the backward pass.

> As is standard with [backpropagation through time] , the network is unfolded over time, so that connections arriving at layers are viewed as coming from the previous timestep.
>
> — *Framewise phoneme classification with bidirectional LSTM and other neural network architectures*, 2005.

Importantly, the backpropagation of error for a given time step depends on the activation of the network at the prior time step. In this way, the backward pass requires the conceptualization of unfolding the network. Error is propagated back to the first input time step of the sequence so that the error gradient can be calculated and the weights of the network can be updated.

> Like standard backpropagation, [backpropagation through time] consists of a repeated application of the chain rule. The subtlety is that, for recurrent networks, the loss function depends on the activation of the hidden layer not only through its influence on the output layer, but also through its influence on the hidden layer at the next timestep.
>
> — *Supervised Sequence Labelling with Recurrent Neural Networks*, 2008.

Unfolding the recurrent network graph also introduces additional concerns. Each time step requires a new copy of the network which in turn takes more memory, especially for large networks with thousands or millions of weights. The memory requirements of training large recurrent networks can quickly balloon as the number of time steps climbs into the hundreds.

> ... it is required to unroll the RNNs by the length of the input sequence. By unrolling an RNN N times, every activations of the neurons inside the network are replicated N times, which consumes a huge amount of memory especially when the sequence is very long. This hinders a small footprint implementation of online learning or adaptation. Also, this "full unrolling" makes a parallel training with multiple sequences inefficient on shared memory models such as graphics processing units (GPUs)
>
> — *Online Sequence Training of Recurrent Neural Networks with Connectionist Temporal Classification*, 2015.

2.3 Backpropagation Through Time

Backpropagation Through Time, or BPTT, is the application of the Backpropagation training algorithm to Recurrent Neural Networks. In the simplest case, a recurrent neural network is shown one input each time step and predicts one output.

Conceptually, BPTT works by unrolling all input time steps. Each time step has one input time step, one copy of the network, and one output. Errors are then calculated and accumulated for each time step. The network is rolled back up and the weights are updated. We can summarize the algorithm as follows:

1. Present a sequence of time steps of input and output pairs to the network.
2. Unroll the network then calculate and accumulate errors across each time step.
3. Roll-up the network and update weights.
4. Repeat.

BPTT can be computationally expensive as the number of time steps increases. If input sequences are comprised of thousands of time steps, then this will be the number of derivatives required for a single weight update. This can cause weights to vanish or explode (go to zero or overflow) and make slow learning and model skill noisy.

> One of the main problems of BPTT is the high cost of a single parameter update, which makes it impossible to use a large number of iterations.
>
> — *Training Recurrent Neural Networks*, 2013

One way to minimize the exploding and vanishing gradient issue is to limit how many time steps before an update to the weights is performed.

2.4 Truncated Backpropagation Through Time

Truncated Backpropagation Through Time, or TBPTT, is a modified version of the BPTT training algorithm for recurrent neural networks where the sequence is processed one time step at a time and periodically an update is performed back for a fixed number of time steps.

> Truncated BPTT ... processes the sequence one time step at a time, and every `k1` time steps, it runs BPTT for `k2` time steps, so a parameter update can be cheap if `k2` is small. Consequently, its hidden states have been exposed to many time steps and so may contain useful information about the far past, which would be opportunistically exploited.
>
> — *Training Recurrent Neural Networks*, 2013

We can summarize the algorithm as follows:

1. Present a sequence of `k1` time steps of input and output pairs to the network.
2. Unroll the network, then calculate and accumulate errors across `k2` time steps.
3. Roll-up the network and update weights.
4. Repeat

The TBPTT algorithm requires the consideration of two parameters:

- `k1`: The number of forward-pass time steps between updates. Generally, this influences how slow or fast training will be, given how often weight updates are performed.
- `k2`: The number of time steps to which to apply BPTT. Generally, it should be large enough to capture the temporal structure in the problem for the network to learn. Too large a value results in vanishing gradients.

A simple implementation of Truncated BPTT would set `k1` to be the sequence length and tune `k2` for both speed of training and model skill.

2.5 Configurations for Truncated BPTT

We can take things one step further and define a notation to help better understand BPTT. In their treatment of BPTT titled *An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories* Williams and Peng devise a notation to capture the spectrum of truncated and untruncated configurations, e.g. `BPTT(h)` and `BPTT(h; 1)`.

We can adapt this notation and use Sutskever's `k1` and `k2` parameters (above). Using this notation, we can define some standard or common approaches: Note: here `n` refers to the total number of time steps in the input sequence:

- `TBPTT(n,n)`: Updates are performed at the end of the sequence across all time steps in the sequence (e.g. classical BPTT).

- `TBPTT(1,n)`: time steps are processed one at a time followed by an update that covers all time steps seen so far (e.g. classical TBPTT by Williams and Peng).
- `TBPTT(k1,1)`: The network likely does not have enough temporal context to learn, relying heavily on internal state and inputs.
- `TBPTT(k1,k2)`, where `k1<k2<n`: Multiple updates are performed per sequence which can accelerate training.
- `TBPTT(k1,k2)`, where `k1=k2`: A common configuration where a fixed number of time steps is used for both forward and backward-pass time steps (e.g. 10s to 100s).

We can see that all configurations are a variation on `TBPTT(n,n)` that essentially attempt to approximate the same effect with perhaps faster training and more stable results. Canonical TBPTT reported in papers may be considered `TBPTT(k1,k2)`, where `k1=k2=k` and `k<=n`, and where the chosen parameter is small (tens to hundreds of time steps). Here, `k` is a single parameter that you must specify. It is often claimed that the sequence length of input time steps should be limited to 200-400.

2.6 Keras Implementation of TBPTT

The Keras deep learning library provides an implementation of TBPTT for training recurrent neural networks. The implementation is more restricted than the general version listed above. Specifically, the `k1` and `k2` values are equal to each other and fixed.

- `TBPTT(k1, k2)`, where `k1=k2=k`.

This is realized by the fixed-sized three-dimensional input required to train recurrent neural networks like the LSTM. The LSTM expects input data to have the dimensions: samples, time steps, and features. It is the second dimension of this input format, the time steps, that defines the number of time steps used for forward and backward passes on your sequence prediction problem.

Therefore, careful choice must be given to the number of time steps specified when preparing your input data for sequence prediction problems in Keras. The choice of time steps will influence both:

- The internal state accumulated during the forward pass.
- The gradient estimate used to update weights on the backward pass.

Note that by default, the internal state of the network is reset after each batch, but more explicit control over when the internal state is reset can be achieved by using a so-called stateful LSTM and calling the reset operation manually. More on this later.

The Keras implementation of the algorithm is essentially un-truncated, requiring that any truncation is performed to the input sequences directly prior to training the model. We can think of this as manually truncated BPTT. Sutskever calls this a *naive* method.

> ... a naive method that splits the 1,000-long sequence into 50 sequences (say) each of length 20 and treats each sequence of length 20 as a separate training case. This is a sensible approach that can work well in practice, but it is blind to temporal dependencies that span more than 20 time steps.
>
> — *Training Recurrent Neural Networks*, 2013

This means as part of framing your problem you must split long sequences into subsequences that are both long enough to capture relevant context for making predictions, but short enough to efficiently train the network.

2.7 Further Reading

This section provides some resources for further reading.

2.7.1 Books

- *Neural Smithing*, 1999.
  http://amzn.to/2u9yjJh
- *Deep Learning*, 2016.
  http://amzn.to/2sx7oFo
- *Supervised Sequence Labelling with Recurrent Neural Networks*, 2008.
  http://amzn.to/2upsSJ9

2.7.2 Research Papers

- *Online Sequence Training of Recurrent Neural Networks with Connectionist Temporal Classification*, 2015.
  https://arxiv.org/abs/1511.06841
- *Framewise phoneme classification with bidirectional LSTM and other neural network architectures*, 2005.
- *Deep learning*, Nature, 2015.
- *Training Recurrent Neural Networks*, 2013.
- *Learning Representations By Backpropagating Errors*, 1986.
- *Backpropagation Through Time: What It Does And How To Do It*, 1990.
- *An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories*, 1990.
- *Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity*, 1995.

2.8 Extensions

Do you want to go deeper into the BPTT algorithm? This section lists some challenging extensions to this lesson.

- Write a one paragraph summary of the BPTT algorithm for a novice practitioner.
- Catalog the BPTT implementation used in top deep learning libraries using the above notation.
- Research and describe the BPTT parameters used in recent or notable LSTM research papers using the above notation.
- Design an experiment to tune the parameters of BPTT for a sequence prediction problem.
- Research and implement the BPTT algorithm for a single memory cell in a spreadsheet or in Python.

Post your extensions online and share the link with me. I'd love to see what you come up with!

2.9 Summary

In this lesson, you discovered the Backpropagation Through Time algorithm used to train LSTMs on sequence prediction problems. Specifically, you learned:

- What Backpropagation Through Time is and how it relates to the Backpropagation training algorithm used by Multilayer Perceptron networks.
- The motivations that lead to the need for Truncated Backpropagation Through Time, the most widely used variant in deep learning for training LSTMs.
- A notation for thinking about how to configure Truncated Backpropagation Through Time and the canonical configurations used in research and by deep learning libraries.

Next, you will discover how to prepare your sequence data for working with LSTMs.

Chapter 3

How to Prepare Data for LSTMs

3.0.1 Lesson Goal

The goal of this lesson is to teach you how to prepare sequence prediction data for use with LSTM models. After completing this lesson, you will know:

- How to scale numeric data and how to transform categorical data.
- How to pad and truncate input sequences with varied lengths.
- How to transform input sequences into a supervised learning problem.

3.0.2 Lesson Overview

This lesson is divided into 4 parts; they are:

1. Prepare Numeric Data.
2. Prepare Categorical Data.
3. Prepare Sequences with Varied Lengths.
4. Sequence Prediction as Supervised Learning.

Let's get started.

3.1 Prepare Numeric Data

The data for your sequence prediction problem probably needs to be scaled when training a neural network, such as a Long Short-Term Memory recurrent neural network. When a network is fit on unscaled data that has a range of values (e.g. quantities in the 10s to 100s) it is possible for large inputs to slow down the learning and convergence of your network, and in some cases prevent the network from effectively learning your problem.

There are two types of scaling of your series that you may want to consider: normalization and standardization. These can both be achieved using the scikit-learn machine learning library in Python.

3.1.1 Normalize Series Data

Normalization is a rescaling of the data from the original range so that all values are within the range of 0 and 1. Normalization requires that you know or are able to accurately estimate the minimum and maximum observable values. You may be able to estimate these values from your available data. If your series is trending up or down, estimating these expected values may be difficult and normalization may not be the best method to use on your problem.

If a value to be scaled is outside the bounds of the minimum and maximum values, the resulting value will not be in the range of 0 and 1. You could check for these observations prior to making predictions and either remove them from the dataset or limit them to the pre-defined maximum or minimum values. You can normalize your dataset using the scikit-learn object `MinMaxScaler`. Good practice usage with the `MinMaxScaler` and other scaling techniques is as follows:

- **Fit the scaler using available training data**. For normalization, this means the training data will be used to estimate the minimum and maximum observable values. This is done by calling the `fit()` function.
- **Apply the scale to training data**. This means you can use the normalized data to train your model. This is done by calling the `transform()` function.
- **Apply the scale to data going forward**. This means you can prepare new data in the future on which you want to make predictions.

If needed, the transform can be inverted. This is useful for converting predictions back into their original scale for reporting or plotting. This can be done by calling the `inverse_transform()` function. Below is an example of normalizing a contrived sequence of 10 quantities. The scaler object requires data to be provided as a matrix of rows and columns. The loaded time series data is loaded as a Pandas `Series`.

```
from pandas import Series
from sklearn.preprocessing import MinMaxScaler
# define contrived series
data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
series = Series(data)
print(series)
# prepare data for normalization
values = series.values
values = values.reshape((len(values), 1))
# train the normalization
scaler = MinMaxScaler(feature_range=(0, 1))
scaler = scaler.fit(values)
print('Min: %f, Max: %f' % (scaler.data_min_, scaler.data_max_))
# normalize the dataset and print
normalized = scaler.transform(values)
print(normalized)
# inverse transform and print
inversed = scaler.inverse_transform(normalized)
print(inversed)
```

Listing 3.1: Example of normalizing a sequence.

Running the example prints the sequence, prints the min and max values estimated from the sequence, prints the same normalized sequence, then prints the values back in their original scale using the inverse transform. We can also see that the minimum and maximum values of the dataset are 10.0 and 100.0 respectively.

```
0    10.0
1    20.0
2    30.0
3    40.0
4    50.0
5    60.0
6    70.0
7    80.0
8    90.0
9   100.0

Min: 10.000000, Max: 100.000000

[[ 0.       ]
 [ 0.11111111]
 [ 0.22222222]
 [ 0.33333333]
 [ 0.44444444]
 [ 0.55555556]
 [ 0.66666667]
 [ 0.77777778]
 [ 0.88888889]
 [ 1.       ]]

[[  10.]
 [  20.]
 [  30.]
 [  40.]
 [  50.]
 [  60.]
 [  70.]
 [  80.]
 [  90.]
 [ 100.]]
```

Listing 3.2: Example output from normalizing a sequence.

3.1.2 Standardize Series Data

Standardizing a dataset involves rescaling the distribution of values so that the mean of observed values is 0 and the standard deviation is 1. This can be thought of as subtracting the mean value or centering the data.

Like normalization, standardization can be useful, and even required in some machine learning algorithms when your data has input values with differing scales. Standardization assumes that your observations fit a Gaussian distribution (bell curve) with a well behaved mean and standard deviation. You can still standardize your time series data if this expectation is not met, but you may not get reliable results.

Standardization requires that you know or are able to accurately estimate the mean and standard deviation of observable values. You may be able to estimate these values from your training data. The mean and standard deviation estimates of a dataset can be more robust to new data than the minimum and maximum. You can standardize your dataset using the scikit-learn object `StandardScaler`.

```
from pandas import Series
from sklearn.preprocessing import StandardScaler
from math import sqrt
# define contrived series
data = [1.0, 5.5, 9.0, 2.6, 8.8, 3.0, 4.1, 7.9, 6.3]
series = Series(data)
print(series)
# prepare data for normalization
values = series.values
values = values.reshape((len(values), 1))
# train the normalization
scaler = StandardScaler()
scaler = scaler.fit(values)
print('Mean: %f, StandardDeviation: %f' % (scaler.mean_, sqrt(scaler.var_)))
# normalize the dataset and print
standardized = scaler.transform(values)
print(standardized)
# inverse transform and print
inversed = scaler.inverse_transform(standardized)
print(inversed)
```

Listing 3.3: Example of standardizing a sequence.

Running the example prints the sequence, prints the mean and standard deviation estimated from the sequence, prints the standardized values, then prints the values back in their original scale. We can see that the estimated mean and standard deviation were about 5.3 and 2.7 respectively.

```
0    1.0
1    5.5
2    9.0
3    2.6
4    8.8
5    3.0
6    4.1
7    7.9
8    6.3

Mean: 5.355556, StandardDeviation: 2.712568

[[-1.60569456]
 [ 0.05325007]
 [ 1.34354035]
 [-1.01584758]
 [ 1.26980948]
 [-0.86838584]
 [-0.46286604]
 [ 0.93802055]
 [ 0.34817357]]
```

```
[[ 1. ]
 [ 5.5]
 [ 9. ]
 [ 2.6]
 [ 8.8]
 [ 3. ]
 [ 4.1]
 [ 7.9]
 [ 6.3]]
```

Listing 3.4: Example output from standardizing a sequence.

3.1.3 Practical Considerations When Scaling

There are some practical considerations when scaling sequence data.

- **Estimate Coefficients**. You can estimate coefficients (min and max values for normalization or mean and standard deviation for standardization) from the training data. Inspect these first-cut estimates and use domain knowledge or domain experts to help improve these estimates so that they will be usefully correct on all data in the future.
- **Save Coefficients**. You will need to scale new data in the future in exactly the same way as the data used to train your model. Save the coefficients used to file and load them later when you need to scale new data when making predictions.
- **Data Analysis**. Use data analysis to help you better understand your data. For example, a simple histogram can help you quickly get a feeling for the distribution of quantities to see if standardization would make sense.
- **Scale Each Series**. If your problem has multiple series, treat each as a separate variable and in turn scale them separately. Here, scale refers a choice of scaling procedure such as normalization or standardization.
- **Scale At The Right Time**. It is important to apply any scaling transforms at the right time. For example, if you have a series of quantities that is non-stationary, it may be appropriate to scale after first making your data stationary. It would not be appropriate to scale the series after it has been transformed into a supervised learning problem as each column would be handled differently, which would be incorrect.
- **Scale if in Doubt**. You probably do need to rescale your input and output variables. If in doubt, at least normalize your data.

3.2 Prepare Categorical Data

Categorical data are variables that contain label values rather than numeric values. The number of possible values is often limited to a fixed set. Categorical variables are often called nominal. Some examples include:

- A `pet` variable with the values: `dog` and `cat`.

- A `color` variable with the values: `red`, `green`, and `blue`.
- A `place` variable with the values: `first`, `second`, and `third`.

Each value represents a different category. Words in text may be considered categorical data, where each word is considered a different category. Additionally, each letter in text data may be considered a category. Sequence prediction problems with text input or output may be considered categorical data.

Some categories may have a natural relationship to each other, such as a natural ordering. The `place` variable above does have a natural ordering of values. This type of categorical variable is called an ordinal variable. Categorical data must be converted to numbers when working with LSTMs.

3.2.1 How to Convert Categorical Data to Numerical Data

This involves two steps:

1. Integer Encoding.
2. One Hot Encoding.

Integer Encoding

As a first step, each unique category value is assigned an integer value. For example, `red` is 1, `green` is 2, and `blue` is 3. This is called label encoding or an integer encoding and is easily reversible. For some variables, this may be enough.

The integer values have a natural ordered relationship between each other and machine learning algorithms may be able to understand and harness this relationship. For example, ordinal variables like the `place` example above would be a good example where a label encoding would be sufficient.

One Hot Encoding

For categorical variables where no such ordinal relationship exists, the integer encoding is not enough. In fact, using this encoding and allowing the model to assume a natural ordering between categories may result in poor performance or unexpected results (predictions halfway between categories).

In this case, a one hot encoding can be applied to the integer representation. This is where the integer encoded variable is removed and a new binary variable is added for each unique integer value. In the `color` variable example, there are 3 categories and therefore 3 binary variables are needed. A `1` value is placed in the binary variable for the color and `0` values for the other colors. For example:

```
red, green, blue
1,   0,    0
0,   1,    0
0,   0,    1
```

Listing 3.5: Example of one hot encoded color.

3.2.2 One Hot Encode with scikit-learn

In this example, we will assume the case where you have an output sequence of the following 3 labels: `cold`, `warm`, `hot`. An example sequence of 10 time steps may be:

```
cold, cold, warm, cold, hot, hot, warm, cold, warm, hot
```

Listing 3.6: Example of categorical temperature sequence.

This would first require an integer encoding, such as 1, 2, 3. This would be followed by a one hot encoding of integers to a binary vector with 3 values, such as `[1, 0, 0]`. The sequence provides at least one example of every possible value in the sequence. Therefore we can use automatic methods to define the mapping of labels to integers and integers to binary vectors.

In this example, we will use the encoders from the scikit-learn library. Specifically, the `LabelEncoder` of creating an integer encoding of labels and the `OneHotEncoder` for creating a one hot encoding of integer encoded values. The complete example is listed below.

```
from numpy import array
from numpy import argmax
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
# define example
data = ['cold', 'cold', 'warm', 'cold', 'hot', 'hot', 'warm', 'cold', 'warm', 'hot']
values = array(data)
print(values)
# integer encode
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(values)
print(integer_encoded)
# binary encode
onehot_encoder = OneHotEncoder(sparse=False, categories='auto')
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
onehot_encoded = onehot_encoder.fit_transform(integer_encoded)
print(onehot_encoded)
# invert first example
inverted = label_encoder.inverse_transform([argmax(onehot_encoded[0, :])])
print(inverted)
```

Listing 3.7: Example of one hot encoding a sequence.

Running the example first prints the sequence of labels. This is followed by the integer encoding of the labels, and finally the one hot encoding. The training data contained the set of all possible examples so we could rely on the integer and one hot encoding transforms to create a complete mapping of labels to encodings.

By default, the `OneHotEncoder` class will return a more efficient sparse encoding. This may not be suitable for some applications, such as use with the Keras deep learning library. In this case, we disabled the sparse return type by setting the `sparse=False` argument. If we receive a prediction in this 3-value one hot encoding, we can easily invert the transform back to the original label.

First, we can use the `argmax()` NumPy function to locate the index of the column with the largest value. This can then be fed to the `LabelEncoder` to calculate an inverse transform back to a text label. This is demonstrated at the end of the example with the inverse transform of the first one hot encoded example back to the label value `cold`. Again, note that input was formatted for readability.

```
['cold' 'cold' 'warm' 'cold' 'hot' 'hot' 'warm' 'cold' 'warm' 'hot']

[0 0 2 0 1 1 2 0 2 1]

[[ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  1.  0.]]

['cold']
```

Listing 3.8: Example output of one hot encoded color.

3.3 Prepare Sequences with Varied Lengths

Deep learning libraries assume a vectorized representation of your data. In the case of variable length sequence prediction problems, this requires that your data be transformed such that each sequence has the same length. This vectorization allows code to efficiently perform the matrix operations in batch for your chosen deep learning algorithms.

3.3.1 Sequence Padding

The `pad_sequences()` function in the Keras deep learning library can be used to pad variable length sequences. The default padding value is `0.0`, which is suitable for most applications, although this can be changed by specifying the preferred value via the `value` argument. For example: The padding to be applied to the beginning or the end of the sequence, called pre- or post-sequence padding, can be specified by the `padding` argument, as follows.

Pre-Sequence Padding

Pre-sequence padding is the default (`padding='pre'`) The example below demonstrates pre-padding 3-input sequences with 0 values.

```
from keras.preprocessing.sequence import pad_sequences
# define sequences
sequences = [
  [1, 2, 3, 4],
     [1, 2, 3],
        [1]
  ]
# pad sequence
padded = pad_sequences(sequences)
print(padded)
```

Listing 3.9: Example of pre-sequence padding.

Running the example prints the 3 sequences pre-pended with zero values.

```
[[1 2 3 4]
[0 1 2 3]
[0 0 0 1]
```

Listing 3.10: Example output of pre-sequence padding.

Post-Sequence Padding

Padding can also be applied to the end of the sequences, which may be more appropriate for some problem domains. Post-sequence padding can be specified by setting the `padding` argument to `post`.

```
from keras.preprocessing.sequence import pad_sequences
# define sequences
sequences = [
  [1, 2, 3, 4],
     [1, 2, 3],
        [1]
  ]
# pad sequence
padded = pad_sequences(sequences, padding='post')
print(padded)
```

Listing 3.11: Example of post-sequence padding.

Running the example prints the same sequences with zero-values appended.

```
[[1 2 3 4]
[1 2 3 0]
[1 0 0 0]]
```

Listing 3.12: Example output of post-sequence padding.

3.3.2 Sequence Truncation

The length of sequences can also be trimmed to a desired length. The desired length for sequences can be specified as a number of time steps with the `maxlen` argument. There are two ways that sequences can be truncated: by removing time steps either from the beginning or the end of sequences.

Pre-Sequence Truncation

The default truncation method is to remove time steps from the beginning of sequences. This is called pre-sequence truncation. The example below truncates sequences to a desired length of 2.

```
from keras.preprocessing.sequence import pad_sequences
# define sequences
sequences = [
  [1, 2, 3, 4],
     [1, 2, 3],
        [1]
  ]
```

```
# truncate sequence
truncated= pad_sequences(sequences, maxlen=2)
print(truncated)
```

Listing 3.13: Example of pre-sequence truncating.

Running the example removes the first two time steps from the first sequence, the first time step from the second sequence, and pads the final sequence.

```
[[3 4]
[2 3]
[0 1]]
```

Listing 3.14: Example output of pre-sequence truncating.

Post-Sequence Truncation

Sequences can also be trimmed by removing time steps from the end of the sequences. This approach may be more desirable for some problem domains. Post-sequence truncation can be configured by changing the `truncating` argument from the default `pre` to `post` as follows:

```
from keras.preprocessing.sequence import pad_sequences
# define sequences
sequences = [
  [1, 2, 3, 4],
     [1, 2, 3],
        [1]
  ]
# truncate sequence
truncated= pad_sequences(sequences, maxlen=2, truncating='post')
print(truncated)
```

Listing 3.15: Example of post-sequence truncating.

Running the example removes the last two time steps from the first sequence, the last time step from the second sequence, and again pads the final sequence.

```
[[1 2]
[1 2]
[0 1]]
```

Listing 3.16: Example output of post-sequence truncating.

There is no rule of thumb as to when to pad and when to truncate input sequences with varied lengths. For example, it may make sense to truncate very long text in a sentiment analysis for efficiency, or it may make sense to pad short text and let the model learn to ignore or explicitly mask zero input values to ensure no data is lost. I recommend testing a suite of different representations for your sequence prediction problem and double down on those that result in the best model skill.

3.4 Sequence Prediction as Supervised Learning

Sequence prediction problems must be re-framed as supervised learning problems. That is, data must be transformed from a sequence to pairs of input and output pairs.

3.4.1 Sequence vs Supervised Learning

Before we get started, let's take a moment to better understand the form of raw input sequence and supervised learning data. Consider a sequence of numbers that are ordered by a time index. This can be thought of as a list or column of ordered values. For example:

```
0
1
2
3
4
5
6
7
8
9
```

Listing 3.17: Example of a sequence.

A supervised learning problem is comprised of input patterns (`X`) and output patterns (`y`), such that an algorithm can learn how to predict the output patterns from the input patterns. For example:

```
X, y
1, 2
2, 3
3, 4
4, 5
5, 6
6, 7
7, 8
8, 9
```

Listing 3.18: Example of input output pairs of a supervised learning problem.

This would represent a 1-lag transform of the sequence, such that the current time step must be predicted given one prior time step of the sequence.

3.4.2 Pandas `shift()` Function

A key function to help transform time series data into a supervised learning problem is the Pandas `shift()` function. Given a `DataFrame`, the `shift()` function can be used to create copies of columns that are pushed forward (rows of `NaN` values added to the front) or pulled back (rows of `NaN` values added to the end).

This is the behavior required to create columns of lag observations as well as columns of forecast observations for a time series dataset in a supervised learning format. Let's look at some examples of the `shift()` function in action. We can define a mock time series dataset as a sequence of 10 numbers, in this case a single column in a `DataFrame` as follows:

```
from pandas import DataFrame
# define the sequence
df = DataFrame()
df['t'] = [x for x in range(10)]
print(df)
```

Listing 3.19: Example of creating a series and printing it.

Running the example prints the time series data with the row indices for each observation.

```
   t
0  0
1  1
2  2
3  3
4  4
5  5
6  6
7  7
8  8
9  9
```

Listing 3.20: Example output of the created series.

We can shift all the observations down by one time step by inserting one new row at the top. Because the new row has no data, we can use `NaN` to represent *no data*. The shift function can do this for us and we can insert this shifted column next to our original series.

```
from pandas import DataFrame
# define the sequence
df = DataFrame()
df['t'] = [x for x in range(10)]
# shift forward
df['t-1'] = df['t'].shift(1)
print(df)
```

Listing 3.21: Example of shifting the series forward.

Running the example gives us two columns in the dataset. The first with the original observations and a new shifted column. We can see that shifting the series forward one time step gives us a primitive supervised learning problem, although with `X` and `y` in the wrong order. Ignore the column of row labels. The first row would have to be discarded because of the `NaN` value. The second row shows the input value of 0.0 in the second column (input or `X`) and the value of 1 in the first column (output or `y`).

```
   t  t-1
0  0  NaN
1  1  0.0
2  2  1.0
3  3  2.0
4  4  3.0
5  5  4.0
6  6  5.0
7  7  6.0
8  8  7.0
9  9  8.0
```

Listing 3.22: Example output of shifting the series forward.

We can see that if we can repeat this process with shifts of 2, 3, and more, we could create long input sequences (`X`) that can be used to forecast an output value (`y`).

The shift operator can also accept a negative integer value. This has the effect of pulling the observations up by inserting new rows at the end. Below is an example:

```
from pandas import DataFrame
# define the sequence
df = DataFrame()
df['t'] = [x for x in range(10)]
# shift backward
df['t+1'] = df['t'].shift(-1)
print(df)
```

Listing 3.23: Example of shifting the series backward.

Running the example shows a new column with a `NaN` value as the last value. We can see that the forecast column can be taken as an input (`X`) and the second as an output value (`y`). That is the input value of 0 can be used to forecast the output value of 1.

```
   t  t+1
0  0  1.0
1  1  2.0
2  2  3.0
3  3  4.0
4  4  5.0
5  5  6.0
6  6  7.0
7  7  8.0
8  8  9.0
9  9  NaN
```

Listing 3.24: Example output of shifting the series backward.

Technically, in time series forecasting terminology the current time (`t`) and future times (`t+1`, `t+n`) are forecast times and past observations (`t-1`, `t-n`) are used to make forecasts. We can see how positive and negative shifts can be used to create a new `DataFrame` from a time series with sequences of input and output patterns for a supervised learning problem.

This permits not only classical `X -> y` prediction, but also `X -> Y` where both input and output can be sequences. Further, the shift function also works on so-called multivariate time series problems. That is where instead of having one set of observations for a time series, we have multiple (e.g. temperature and pressure). All variates in the time series can be shifted forward or backward to create multivariate input and output sequences.

3.5 Further Reading

This section provides some resources for further reading.

3.5.1 Numeric Scaling APIs

- `MinMaxScaler` API in scikit-learn.
  https://goo.gl/H3qHJU
- `StandardScaler` API in scikit-learn.
  https://goo.gl/cA4vQi

- Should I normalize/standardize/rescale the data? Neural Nets FAQ.
  ftp://ftp.sas.com/pub/neural/FAQ2.html#A_std

3.5.2 Categorical Encoding APIs

- `LabelEncoder` API in scikit-learn.
  https://goo.gl/Y2bn3T
- `OneHotEncoder` API in scikit-learn.
  https://goo.gl/ynDMHN
- NumPy `argmax()` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html

3.5.3 Varied Length Sequence APIs

- `pad_sequences()` API in Keras.
  https://keras.io/preprocessing/sequence/

3.5.4 Supervised Learning APIs

- `shift()` function API in Pandas.
  https://goo.gl/N3M3nG

3.6 Extensions

Do you want to go deeper into data preparation for LSTMs? This section lists some challenging extensions to this lesson.

- In a paragraph, summarize when to normalize numeric data, when to standardize, and what to do when you're in doubt.
- Develop a function to automatically one hot encode ASCII text as categorical data, and another function to decode the encoded format.
- List 3 examples of sequence prediction problems that may benefit from padding input sequences and 3 that may benefit from truncating input sequences.
- Given a univariate time series forecasting problem with hourly observations over several years, and given an understanding of truncated BPTT in the previous lesson, describe 3 ways that the sequence may be transformed into a supervised learning problem.
- Develop a Python function to automatically transform a series into a supervised learning problem where the number of input and output time steps can be specified as arguments.

Post your extensions online and share the link with me; I'd love to see what you come up with!

3.7 Summary

In this lesson, you discovered how to prepare sequence data for working with LSTM recurrent neural networks. Specifically, you learned:

- How to scale numeric data and how to transform categorical data.
- How to pad and truncate input sequences with varied lengths.
- How to transform input sequences into a supervised learning problem.

Next, you will discover the life-cycle of an LSTM model in the Keras library.

Chapter 4

How to Develop LSTMs in Keras

4.0.1 Lesson Goal

The goal of this lesson is to understand how to define, fit, and evaluate LSTM models using the Keras deep learning library in Python. After completing this lesson, you will know:

- How to define an LSTM model, including how to reshape your data for the required 3D input.
- How to fit and evaluate your LSTM model and use it to make predictions on new data.
- How to take fine-grained control over the internal state in the model and when it is reset.

4.0.2 Lesson Overview

This lesson is divided into 7 parts; they are:

1. Define the Model.
2. Compile the Model.
3. Fit the Model.
4. Evaluate the Model.
5. Make Predictions with the Model.
6. LSTM State Management.
7. Examples of Preparing Data.

Let's get started.

Note: The Keras code examples in this chapter are demonstrations to familiarize you with the API, they do not execute.

4.1 Define the Model

The first step is to define your network. Neural networks are defined in Keras as a sequence of layers. The container for these layers is the `Sequential` class. The first step is to create an instance of the `Sequential` class. Then you can create your layers and add them in the order that they should be connected. The LSTM recurrent layer comprised of memory units is called `LSTM()`. A fully connected layer that often follows LSTM layers and is used for outputting a prediction is called `Dense()`.

For example, we can define an `LSTM` hidden layer with 2 memory cells followed by a `Dense` output layer with 1 neuron as follows:

```
model = Sequential()
model.add(LSTM(2))
model.add(Dense(1))
```

Listing 4.1: Example of defining an LSTM model.

But we can also do this in one step by creating an array of layers and passing it to the constructor of the `Sequential` class.

```
layers = [LSTM(2), Dense(1)]
model = Sequential(layers)
```

Listing 4.2: A second example of defining an LSTM model.

The first hidden layer in the network must define the number of inputs to expect, e.g. the shape of the input layer. Input must be three-dimensional, comprised of samples, time steps, and features in that order.

- **Samples**. These are the rows in your data. One sample may be one sequence.
- **Time steps**. These are the past observations for a feature, such as lag variables.
- **Features**. These are columns in your data.

Assuming your data is loaded as a NumPy array, you can convert a 1D or 2D dataset to a 3D dataset using the `reshape()` function in NumPy. You can call the `reshape()` function on your NumPy array and pass it a tuple of the dimensions to which to transform your data. Imagine we had 2 columns of input data (`X`) in a NumPy array. We could treat the two columns as two time steps and reshape it as follows:

```
data = data.reshape((data.shape[0], data.shape[1], 1))
```

Listing 4.3: Example of reshaping a NumPy array with 1 feature.

If you would like columns in your 2D data to become features with one time step, you can reshape it as follows:

```
data = data.reshape((data.shape[0], 1, data.shape[1]))
```

Listing 4.4: Example of reshaping a NumPy array with 1 time step.

You can specify the `input_shape` argument that expects a tuple containing the number of time steps and the number of features. For example, if we had two time steps and one feature for a univariate sequence with two lag observations per row, it would be specified as follows:

```
model = Sequential()
model.add(LSTM(5, input_shape=(2,1)))
model.add(Dense(1))
```

Listing 4.5: Example defining the input shape for an LSTM model.

The number of samples does not have to be specified. The model assumes one or more samples, leaving you to define only the number of time steps and features. The final section of this lesson provides additional examples of preparing input data for LSTM models.

Think of a `Sequential` model as a pipeline with your raw data fed in at one end and predictions that come out at the other. This is a helpful container in Keras as concerns that were traditionally associated with a layer can also be split out and added as separate layers, clearly showing their role in the transform of data from input to prediction. For example, activation functions that transform a summed signal from each neuron in a layer can be extracted and added to the `Sequential` as a layer-like object called `Activation`.

```
model = Sequential()
model.add(LSTM(5, input_shape=(2,1)))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Listing 4.6: Example of an LSTM model with sigmoid activation on the output layer.

The choice of activation function is most important for the output layer as it will define the format that predictions will take. For example, below are some common predictive modeling problem types and the structure and standard activation function that you can use in the output layer:

- **Regression**: Linear activation function, or `linear`, and the number of neurons matching the number of outputs. This is the default activation function used for neurons in the `Dense` layer.
- **Binary Classification (2 class)**: Logistic activation function, or `sigmoid`, and one neuron the output layer.
- **Multiclass Classification (> 2 class)**: Softmax activation function, or `softmax`, and one output neuron per class value, assuming a one hot encoded output pattern.

4.2 Compile the Model

Once we have defined our network, we must compile it. Compilation is an efficiency step. It transforms the simple sequence of layers that we defined into a highly efficient series of matrix transforms in a format intended to be executed on your GPU or CPU, depending on how Keras is configured. Think of compilation as a precompute step for your network. It is always required after defining a model.

Compilation requires a number of parameters to be specified, specifically tailored to training your network. Specifically, the optimization algorithm to use to train the network and the loss function used to evaluate the network that is minimized by the optimization algorithm.

For example, below is a case of compiling a defined model and specifying the stochastic gradient descent (`sgd`) optimization algorithm and the mean squared error (`mse`) loss function, intended for a regression type problem.

```
model.compile(optimizer='sgd', loss='mse')
```

Listing 4.7: Example of compiling an LSTM model.

Alternately, the optimizer can be created and configured before being provided as an argument to the compilation step.

```
algorithm = SGD(lr=0.1, momentum=0.3)
model.compile(optimizer=algorithm, loss='mse')
```

Listing 4.8: Example of compiling an LSTM model with a SGD optimization algorithm.

The type of predictive modeling problem imposes constraints on the type of loss function that can be used. For example, below are some standard loss functions for different predictive model types:

- **Regression**: Mean Squared Error or `mean_squared_error`, `mse` for short.
- **Binary Classification (2 class)**: Logarithmic Loss, also called cross entropy or `binary_crossentropy`.
- **Multiclass Classification (> 2 class)**: Multiclass Logarithmic Loss or `categorical_crossentropy`.

The most common optimization algorithm is classical stochastic gradient descent, but Keras also supports a suite of other extensions of this classic optimization algorithm that work well with little or no configuration. Perhaps the most commonly used optimization algorithms because of their generally better performance are:

- **Stochastic Gradient Descent**, or `sgd`.
- **Adam**, or `adam`.
- **RMSprop**, or `rmsprop`.

Finally, you can also specify metrics to collect while fitting your model in addition to the loss function. Generally, the most useful additional metric to collect is accuracy for classification problems (e.g. `'accuracy'` or `'acc'` for short). The metrics to collect are specified by name in an array of metric or loss function names. For example:

```
model.compile(optimizer='sgd', loss='mean_squared_error', metrics=['accuracy'])
```

Listing 4.9: Example of compiling an LSTM model with a metric.

4.3 Fit the Model

Once the network is compiled, it can be fit, which means adapting the weights on a training dataset. Fitting the network requires the training data to be specified, both a matrix of input patterns, `X`, and an array of matching output patterns, `y`. The network is trained using the Backpropagation Through Time algorithm and optimized according to the optimization algorithm and loss function specified when compiling the model.

The backpropagation algorithm requires that the network be trained for a specified number of epochs or exposures to all sequences in the training dataset. Each epoch can be partitioned into groups of input-output pattern pairs called batches. This defines the number of patterns that the network is exposed to before the weights are updated within an epoch. It is also an efficiency optimization, ensuring that not too many input patterns are loaded into memory at a time.

- **Epoch**: One pass through all samples in the training dataset and updating the network weights. LSTMs may be trained for tens, hundreds, or thousands of epochs.
- **Batch**: A pass through a subset of samples in the training dataset after which the network weights are updated. One epoch is comprised of one or more batches.

Below are some common configurations for the batch size:

- `batch_size=1`: Weights are updated after each sample and the procedure is called stochastic gradient descent.
- `batch_size=32`: Weights are updated after a specified number of samples and the procedure is called mini-batch gradient descent. Common values are 32, 64, and 128, tailored to the desired efficiency and rate of model updates. If the batch size is not a factor of the number of samples in one epoch, then an additional batch size of the left over samples is run at the end of the epoch.
- `batch_size=n`: Where `n` is the number of samples in the training dataset. Weights are updated at the end of each epoch and the procedure is called batch gradient descent.

Mini-batch gradient descent with a batch size of 32 is a common configuration for LSTMs. An example of fitting a network is as follows:

```
model.fit(X, y, batch_size=32, epochs=100)
```

Listing 4.10: Example of fitting an LSTM model.

Once fit, a history object is returned that provides a summary of the performance of the model during training. This includes both the loss and any additional metrics specified when compiling the model, recorded each epoch. These metrics can be recorded, plotted, and analyzed to gain insight into whether the network is overfitting or underfitting the training data.

Training can take a long time, from seconds to hours to days depending on the size of the network and the size of the training data. By default, a progress bar is displayed on the command line for each epoch. This may create too much noise for you, or may cause problems for your environment, such as if you are in an interactive notebook or IDE. You can reduce the amount of information displayed to just the loss each epoch by setting the `verbose` argument to 2. You can turn off all output by setting `verbose` to 0. For example:

```
history = model.fit(X, y, batch_size=10, epochs=100, verbose=0)
```

Listing 4.11: Example of fitting an LSTM model and retrieving history without verbose output.

4.4 Evaluate the Model

Once the network is trained, it can be evaluated. The network can be evaluated on the training data, but this will not provide a useful indication of the performance of the network as a predictive model, as it has seen all of this data before. We can evaluate the performance of the network on a separate dataset, unseen during training. This will provide an estimate of the performance of the network at making predictions for unseen data in the future.

The model evaluates the loss across all of the test patterns, as well as any other metrics specified when the model was compiled, like classification accuracy. A list of evaluation metrics is returned. For example, for a model compiled with the accuracy metric, we could evaluate it on a new dataset as follows:

```
loss, accuracy = model.evaluate(X, y)
```

Listing 4.12: Example of evaluating an LSTM model.

As with fitting the network, verbose output is provided to give an idea of the progress of evaluating the model. We can turn this off by setting the `verbose` argument to 0.

```
loss, accuracy = model.evaluate(X, y, verbose=0)
```

Listing 4.13: Example of evaluating an LSTM model without verbose output.

4.5 Make Predictions on the Model

Once we are satisfied with the performance of our fit model, we can use it to make predictions on new data. This is as easy as calling the `predict()` function on the model with an array of new input patterns. For example:

```
predictions = model.predict(X)
```

Listing 4.14: Example of making a prediction with a fit LSTM model.

The predictions will be returned in the format provided by the output layer of the network. In the case of a regression problem, these predictions may be in the format of the problem directly, provided by a linear activation function.

For a binary classification problem, the predictions may be an array of probabilities for the first class that can be converted to a 1 or 0 by rounding. For a multiclass classification problem, the results may be in the form of an array of probabilities (assuming a one hot encoded output variable) that may need to be converted to a single class output prediction using the `argmax()` NumPy function. Alternately, for classification problems, we can use the `predict_classes()` function that will automatically convert uncrisp predictions to crisp integer class values.

```
predictions = model.predict_classes(X)
```

Listing 4.15: Example of predicting classes with a fit LSTM model.

As with fitting and evaluating the network, verbose output is provided to give an idea of the progress of the model making predictions. We can turn this off by setting the `verbose` argument to 0.

```
predictions = model.predict(X, verbose=0)
```

Listing 4.16: Example of making a prediction without verbose output.

Making predictions with fit LSTM models is covered in more detail in Chapter 13.

4.6 LSTM State Management

Each LSTM memory unit maintains internal state that is accumulated. This internal state may require careful management for your sequence prediction problem both during the training of the network and when making predictions. By default, the internal state of all LSTM memory units in the network is reset after each batch, e.g. when the network weights are updated. This means that the configuration of the batch size imposes a tension between three things:

- The efficiency of learning, or how many samples are processed before an update.
- The speed of learning, or how often weights are updated.
- The influence of internal state, or how often internal state is reset.

Keras provides flexibility to decouple the resetting of internal state from updates to network weights by defining an `LSTM` layer as stateful. This can be done by setting the `stateful` argument on the `LSTM` layer to `True`. When stateful LSTM layers are used, you must also define the batch size as part of the input shape in the definition of the network by setting the `batch_input_shape` argument and the batch size must be a factor of the number of samples in the training dataset. The `batch_input_shape` argument requires a 3-dimensional tuple defined as batch size, time steps, and features.

For example, we can define a stateful LSTM to be trained on a training dataset with 100 samples, a batch size of 10, and 5 time steps for 1 feature, as follows.

```
model.add(LSTM(2, stateful=True, batch_input_shape=(10, 5, 1)))
```

Listing 4.17: Example of defining a stateful `LSTM` layer.

A stateful LSTM will not reset the internal state at the end of each batch. Instead, you have fine grained control over when to reset the internal state by calling the `reset_states()` function. For example, we may want to reset the internal state at the end of each single epoch which we could do as follows:

```
for i in range(1000):
    model.fit(X, y, epochs=1, batch_input_shape=(10, 5, 1))
    model.reset_states()
```

Listing 4.18: Example of manually iterating training epochs for a stateful LSTM.

The same batch size used in the definition of the stateful LSTM must also be used when making predictions.

```
predictions = model.predict(X, batch_size=10)
```

Listing 4.19: Example making predictions with a stateful LSTM.

The internal state in LSTM layers is also accumulated when evaluating a network and when making predictions. Therefore, if you are using a stateful LSTM, you must reset state after evaluating the network on a validation dataset or after making predictions.

By default, the samples within an epoch are shuffled. This is a good practice when working with Multilayer Perceptron neural networks. If you are trying to preserve state across samples, then the order of samples in the training dataset may be important and must be preserved. This can be done by setting the `shuffle` argument in the `fit()` function to `False`. For example:

```
for i in range(1000):
    model.fit(X, y, epochs=1, shuffle=False, batch_input_shape=(10, 5, 1))
    model.reset_states()
```

Listing 4.20: Example disabling sample shuffling when fitting a stateful LSTM.

To make this more concrete, below are a 3 common examples for managing state:

- A prediction is made at the end of each sequence and sequences are independent. State should be reset after each sequence by setting the `batch_size` to 1.
- A long sequence was split into multiple subsequences (many samples each with many time steps). State should be reset after the network has been exposed to the entire sequence by making the LSTM stateful, turning off the shuffling of subsequences, and resetting the state after each epoch.
- A very long sequence was split into multiple subsequences (many samples each with many time steps). Training efficiency is more important than the influence of long-term internal state and a batch size of 128 samples was used, after which network weights are updated and state reset.

I would encourage you to brainstorm many different framings of your sequence prediction problem and network configurations, test and select those models that appear most promising with regard to prediction error.

4.7 Examples of Preparing Data

It can be difficult to understand how to prepare your sequence data for input to an LSTM model. Often there is confusion around how to define the input layer for the LSTM model. There is also confusion about how to convert your sequence data that may be a 1D or 2D matrix of numbers to the required 3D format of the LSTM input layer. In this section you will work through two examples of reshaping sequence data and defining the input layer to LSTM models.

4.7.1 Example of LSTM With Single Input Sample

Consider the case where you have one sequence of multiple time steps and one feature. For example, this could be a sequence of 10 values:

```
0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0
```

Listing 4.21: Example of a sequence.

We can define this sequence of numbers as a NumPy array.

```
from numpy import array
data = array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
```

Listing 4.22: Example of defining a sequence as a NumPy array.

We can then use the `reshape()` function on the NumPy array to reshape this one-dimensional array into a three-dimensional array with 1 sample, 10 time steps and 1 feature at each time step. The `reshape()` function when called on an array takes one argument which is a tuple defining the new shape of the array. We cannot pass in any tuple of numbers, the reshape must evenly reorganize the data in the array.

```
data = data.reshape((1, 10, 1))
```

Listing 4.23: Example of reshaping a sequence.

Once reshaped, we can print the new shape of the array.

```
print(data.shape)
```

Listing 4.24: Example of printing the new shape of the sequence.

Putting all of this together, the complete example is listed below.

```
from numpy import array
data = array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
data = data.reshape((1, 10, 1))
print(data.shape)
```

Listing 4.25: Example of reshaping one sample.

Running the example prints the new 3D shape of the single sample.

```
(1, 10, 1)
```

Listing 4.26: Example output from reshaping one sample.

This data is now ready to be used as input (`X`) to the LSTM with an `input_shape` of `(10, 1)`.

```
model = Sequential()
model.add(LSTM(32, input_shape=(10, 1)))
...
```

Listing 4.27: Example of defining the input layer for an LSTM model.

4.7.2 Example of LSTM With Multiple Input Features

Consider the case where you have multiple parallel series as input for your model. For example, this could be two parallel series of 10 values:

```
series 1: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0
series 2: 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1
```

Listing 4.28: Example of parallel sequences.

We can define these data as a matrix of 2 columns with 10 rows:

```
from numpy import array
data = array([
  [0.1, 1.0],
  [0.2, 0.9],
  [0.3, 0.8],
  [0.4, 0.7],
  [0.5, 0.6],
  [0.6, 0.5],
  [0.7, 0.4],
  [0.8, 0.3],
  [0.9, 0.2],
  [1.0, 0.1]])
```

Listing 4.29: Example of defining parallel sequences as a NumPy array.

This data can be framed as 1 sample with 10 time steps and 2 features. It can be reshaped as a 3D array as follows:

```
data = data.reshape(1, 10, 2)
```

Listing 4.30: Example of reshaping a sequence.

Putting all of this together, the complete example is listed below.

```
from numpy import array
data = array([
  [0.1, 1.0],
  [0.2, 0.9],
  [0.3, 0.8],
  [0.4, 0.7],
  [0.5, 0.6],
  [0.6, 0.5],
  [0.7, 0.4],
  [0.8, 0.3],
  [0.9, 0.2],
  [1.0, 0.1]])
data = data.reshape(1, 10, 2)
print(data.shape)
```

Listing 4.31: Example of reshaping parallel series.

Running the example prints the new 3D shape of the single sample.

```
(1, 10, 2)
```

Listing 4.32: Example output from reshaping parallel series.

This data is now ready to be used as input (`X`) to the LSTM with an `input_shape` of `(10, 2)`.

```
model = Sequential()
model.add(LSTM(32, input_shape=(10, 2)))
...
```

Listing 4.33: Example of defining the input layer for an LSTM model.

4.7.3 Tips for LSTM Input

This section lists some final tips to help you when preparing your input data for LSTMs.

- The LSTM input layer must be 3D.
- The meaning of the 3 input dimensions are: samples, time steps and features.
- The LSTM input layer is defined by the `input_shape` argument on the first hidden layer.
- The `input_shape` argument takes a tuple of two values that define the number of time steps and features.
- The number of samples is assumed to be 1 or more.
- The `reshape()` function on NumPy arrays can be used to reshape your 1D or 2D data to be 3D.
- The `reshape()` function takes a tuple as an argument that defines the new shape.

4.8 Further Reading

This section provides some resources for further reading.

4.8.1 Keras APIs

- Keras API for `Sequential` Models.
  https://keras.io/models/sequential/
- Keras API for `LSTM` Layers.
  https://keras.io/layers/recurrent/#lstm
- Keras API for optimization algorithms.
  https://keras.io/optimizers/
- Keras API for loss functions.
  https://keras.io/losses/

4.8.2 Other APIs

- NumPy `reshape()` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html
- NumPy `argmax()` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html

4.9 Extensions

Do you want to dive deeper into the life-cycle of LSTMs in Keras? This section lists some challenging extensions to this lesson.

- List 5 sequence prediction problems and highlight how the data breaks down into samples, time steps, and features.
- List 5 sequence prediction problems and specify the activation functions used in the output layer for each.
- Research Keras metrics and loss functions and list 5 metrics that can be used for a regression sequence prediction problem and 5 for a classification sequence prediction problem.
- Research the Keras history object and write example code to create a line plot with Matplotlib of the metrics captured from fitting an LSTM model.
- List 5 sequence prediction problems and how you would define and fit a network to best manage the internal state for each.

Post your extensions online and share the link with me. I'd love to see what you come up with!

4.10 Summary

In this lesson, you discovered the 5-step lifecycle of an LSTM recurrent neural network using the Keras library. Specifically, you learned:

- How to define an LSTM model, including how to reshape your data for the required 3D input.
- How to fit and evaluate your LSTM model and use it to make predictions on new data.
- How to take fine-grained control over the internal state in the model and when it is reset.

In the next lesson, you will discover the 4 main types of sequence prediction models and how to implement them in Keras.

Chapter 5

Models for Sequence Prediction

5.0.1 Lesson Goal

The goal of this lesson is for you to know about the 4 sequence prediction models and how to realize them in Keras. After completing this lesson, you will know:

- The 4 models for sequence prediction and how they may be implemented in Keras.
- Examples of how to map the 4 sequence prediction models onto common and interesting sequence prediction problems.
- The traps that beginners fall into in applying the sequence prediction models and how to avoid them.

5.0.2 Lesson Overview

This lesson is divided into 5 parts; they are:

1. Sequence Prediction.
2. Models for Sequence Prediction.
3. Mapping Applications to Models.
4. Cardinality from Time Steps.
5. Two Common Misunderstandings.

Let's get started.

5.1 Sequence Prediction

LSTMs work by learning a function (`f(...)`) that maps input sequence values (`X`) onto output sequence values (`y`).

```
y(t) = f(X(t))
```

Listing 5.1: Example of the general LSTM model.

The learned mapping function is static and may be thought of as a program that takes input variables and uses internal variables. Internal variables are represented by an internal state maintained by the network and built up or accumulated over each value in the input sequence. The static mapping function may be defined with a different number of inputs or outputs. Understanding this important detail is the focus of this lesson.

5.2 Models for Sequence Prediction

In this section, will review the 4 primary models for sequence prediction. We will use the following terminology:

- `X`: The input sequence value; may be delimited by a time step, e.g. `X(1)`.
- `u`: The hidden state value; may be delimited by a time step, e.g. `u(1)`.
- `y`: The output sequence value; may be delimited by a time step, e.g. `y(1)`.

Each model will be explained using this terminology, using pictures, and using example code in Keras. Focus on learning how different types of sequence prediction problems map to different model types. Don't get too caught up on the specifics of the Keras examples, as whole chapters are dedicated to explaining the more complex model types.

5.2.1 One-to-One Model

A one-to-one model (`f(...)`) produces one output (`y(t)`) value for each input value (`X(t)`).

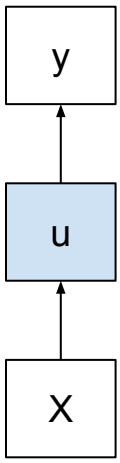

Figure 5.1: One-to-One Sequence Prediction Model.

For example:

```
y(1) = f(X(1))
y(2) = f(X(2))
y(3) = f(X(3))
...
```

Listing 5.2: Example of a one-to-one sequence prediction model.

The internal state for the first time step is zero; from that point onward, the internal state is accumulated over the prior time steps.

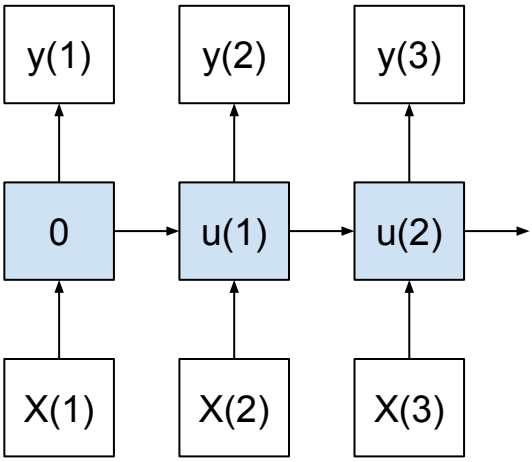

Figure 5.2: One-to-One Sequence Prediction Model Over Time.

This model is appropriate for sequence prediction problems where we wish to predict one time step, given one time step as input. For example:

- Predicting the next real value in a time series.
- Predicting the next word in a sentence.

This is a poor use of the LSTM as it is not capable of learning across input or output time steps. This model does put all of the pressure on the internal state or memory. The results of this model can be contrasted to a model that does have input or output sequences to see if learning across time steps adds skill to predictions. We can implement this in Keras by defining a network that expects one time step as input and predicts one time step in the output layer.

```
model = Sequential()
model.add(LSTM(..., input_shape=(1, ...)))
model.add(Dense(1))
```

Listing 5.3: Example of defining a one-to-one sequence prediction model in Keras.

A one-to-one LSTM model is developed for a sequence generation problem in Chapter 11 on page 140.

5.2.2 One-to-Many Model

A one-to-many model (`f(...)`) produces multiple output values (`y(t)`, `y(t+1)`, ...) for one input value (`X(t)`). For example:

```
y(1),y(2) = f(X(1))
```

Listing 5.4: Example of a one-to-many sequence prediction model.

It may help to think of time being counted separately for inputs and outputs.

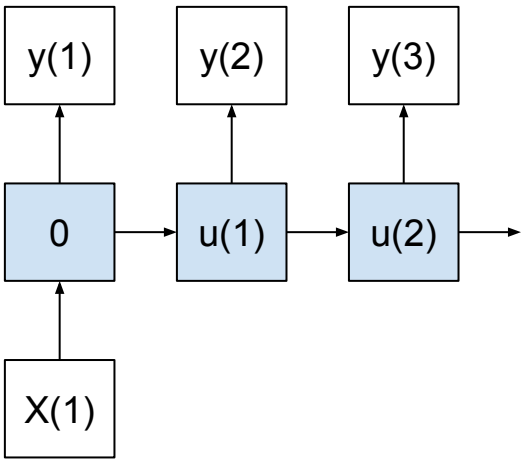

Figure 5.3: One-to-Many Sequence Prediction Model.

Internal state is accumulated as each value in the output sequence is produced. This model is appropriate for sequence prediction problems where we wish to produce a sequence output for each input time step. For example:

- Predicting a sequence of words from a single image.
- Forecasting a series of observations from a single event.

A good example of this model is in generating textual captions for images. In Keras, this requires a Convolutional Neural Network to extract features from the image followed by an LSTM to output the sequence of words one at a time. The output layer predicts one observation per output time step and is wrapped in a `TimeDistributed` wrapper layer in order to use the same output layer multiple times for the required number of output time steps.

```
model = Sequential()
model.add(Conv2D(...))
...
model.add(LSTM(...))
model.add(TimeDistributed(Dense(1)))
```

Listing 5.5: Example of defining a one-to-many sequence prediction model in Keras.

The sequence classification example in Chapter 8 on page 92 could be adapted to a one-to-many model. For example, for describing the up, down, left and right movement of the line from a final image.

5.2.3 Many-to-One Model

A many-to-one model (`f(...)`) produces one output (`y(t)`) value after receiving multiple input values (`X(t)`, `X(t+1)`, ...). For example:

```
y(1) = f(X(1), X(2))
```

Listing 5.6: Example of a many-to-one sequence prediction model.

Again, it helps to think of time being counted separately in the input sequences and the output sequences.

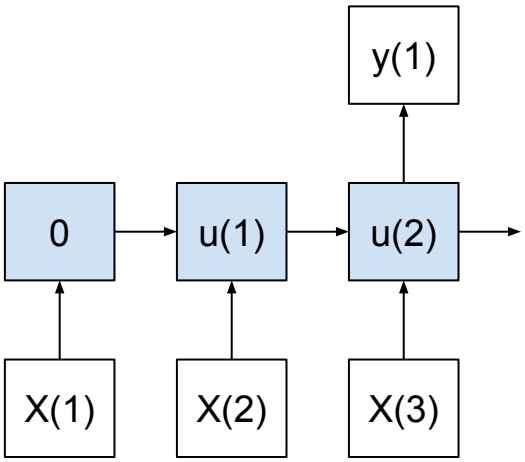

Figure 5.4: Many-to-One Sequence Prediction Model.

Internal state is accumulated with each input value before a final output value is produced. This model is appropriate for sequence prediction problems where multiple input time steps are required in order to make a one step prediction. For example:

- Forecasting the next real value in a time series given a sequence of input observations.
- Predicting the classification label for an input sequence of words, such as sentiment analysis.

This model can be implemented in Keras much like the one-to-one model, except the number of input time steps can be varied to suit the needs of the problem.

```
model = Sequential()
model.add(LSTM(..., input_shape=(steps, ...)))
model.add(Dense(1))
```

Listing 5.7: Example of defining a many-to-one sequence prediction model in Keras.

Often, when practitioners implement a one-to-one model, they actually intend to implement a many-to-one model, such as in the case of time series forecasting. A many-to-one model is developed for sequence classification in Chapter 6 on page 65, for predicting a single vector output in Chapter 7 on page 77 and for sequence classification in Chapter 8 on page 92.

5.2.4 Many-to-Many Model

A many-to-many model (`f(...)`) produces multiple outputs (`y(t)`, `y(t+1)`, ...) after receiving multiple input values (`X(t)`, `X(t+1)`, ...). For example:

```
y(1),y(2) = f(X(1), X(2))
```

Listing 5.8: Example of a many-to-many sequence prediction model.

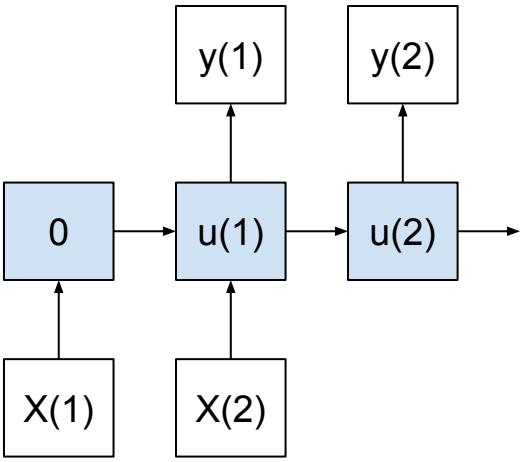

Figure 5.5: Many-to-Many Sequence Prediction Model.

As with the many-to-one case, state is accumulated until the first output is created, but in this case multiple time steps are output. Importantly, the number of input time steps do not have to match the number of output time steps. This model is appropriate for sequence predictions where multiple input time steps are required in order to predict a sequence of output time steps. These are often called sequence-to-sequence, or seq2seq, type problems and are perhaps the most studied with LSTMs in recent time. For example:

- Summarize a document of words into a shorter sequence of words.
- Classify a sequence of audio data into a sequence of words.

In a sense, this model combines the capabilities of the many-to-one and one-to-many models. If the number of input and output time steps are equal, then the `LSTM` layer must be configured to return a value for each input time step rather than a single value at the end of the input sequence (e.g. `return_sequences=True`) and the same `Dense` layer can be used to produce one output time step for each of the input time steps via the `TimeDistributed` layer wrapper

```
model = Sequential()
model.add(LSTM(..., input_shape=(steps, ...), return_sequences=True))
model.add(TimeDistributed(Dense(1)))
```

Listing 5.9: Example of defining a many-to-many sequence prediction model in Keras with equal length input and output sequences.

If the number of input and output time steps vary, then an Encoder-Decoder architecture can be used. The input time steps are mapped to a fixed sized internal representation of the sequence, then this vector is used as input to producing each time step in the output sequence.

```
model = Sequential()
model.add(LSTM(..., input_shape=(in_steps, ...)))
model.add(RepeatVector(out_steps))
model.add(LSTM(..., return_sequences=True))
model.add(TimeDistributed(Dense(1)))
```

Listing 5.10: Example of defining a many-to-many sequence prediction model in Keras with varying length input and output sequences.

This may be the most sophisticated sequence prediction model with many variations and optimizations to explore. A many-to-many model is developed in Chapter 9 on page 107 where input and output sequences have a different number of time steps (e.g. sequence lengths). A many-to-many model is also developed in Chapter 10 on page 128 where the input and output sequences have the same number of time steps.

5.3 Mapping Applications to Models

I really want you to understand these models and how to frame your problem as one of the above 4 types. To that end, this section lists 10 different and varied sequence prediction problems and notes which model may be used to address them. In each explanation, I give an example of a model that *can* be used to address the problem, but other models can be used if the sequence prediction problem is re-framed. Take these as best suggestions, not unbreakable rules.

5.3.1 Time Series

- **Univariate Time Series Forecasting**. This is where you have one series with multiple input time steps and wish to predict one time step beyond the input sequence. This can be implemented as a many-to-one model.
- **Multivariate Time Series Forecasting**. This is where you have multiple series with multiple input time steps and wish to predict one time step beyond one or more of the input sequences. This can be implemented as a many-to-one model. Each series is just another input feature.
- **Multi-step Time Series Forecasting**: This is where you have one or multiple series with multiple input time steps and wish to predict multiple time steps beyond one or more of the input sequences. This can be implemented as a many-to-many model.
- **Time Series Classification**. This is where you have one or multiple series with multiple input time steps as input and wish to output a classification label. This can be implemented as a many-to-one model.

5.3.2 Natural Language Processing

- **Image Captioning**. This is where you have one image and wish to generate a textual description. This can be implemented as a one-to-many model.
- **Video Description**. This is where you have a sequence of images in a video and wish to generate a textual description. This can be implemented with a many-to-many model.

- **Sentiment Analysis**. This is where you have sequences of text as input and you wish to generate a classification label. This can be implemented as a many-to-one model.
- **Speech Recognition**. This is where you have a sequence of audio data as input and wish to generate a textual description of what was spoken. This can be implemented with a many-to-many model.
- **Text Translation**. This is where you have a sequence of words in one language as input and wish to generate a sequence of words in another language. This can be implemented with a many-to-many model.
- **Text Summarization**. This is where you have a document of text as input and wish to create a short textual summary of the document as output. This can be implemented with a many-to-many model.

5.4 Cardinality from Time Steps (not Features!)

A common point of confusion is to conflate the above examples of sequence mapping models with multiple input and output features. A sequence may be comprised of single values, one for each time step.

Alternately, a sequence could just as easily represent a vector of multiple observations at the time step (e.g. `[X1, X2]` to predict `[y1, y2]` at a given time step). Each item in the vector for a time step may be thought of as its own separate time series. It does not affect the description of the models above. For example, a model that takes as input one time step of temperature (`X1`) and pressure (`X2`) and predicts one time step of temperature (`y1`) and pressure (`y2`) is a one-to-one model, not a many-to-many model.

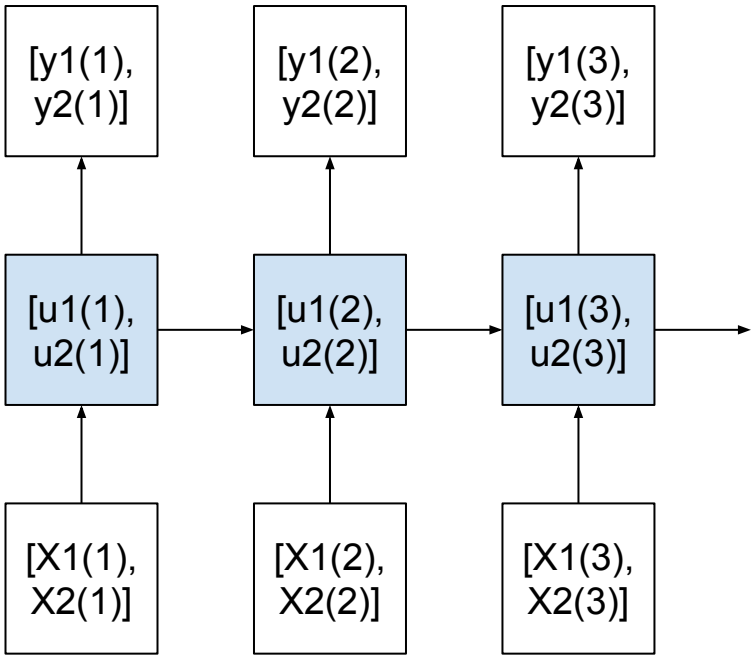

Figure 5.6: Multiple-Feature Sequence Prediction Model.

The model does take two values as input, has two separate internal states, and predicts two values, but there is only a single sequence time step expressed for the input and predicted as output. The cardinality of the sequence prediction models defined above refers to time steps, not features.

5.5 Two Common Misunderstandings

The confusion of features vs time steps leads to two main misunderstandings when implementing LSTMs by practitioners:

5.5.1 Time steps as Input Features

Lag observations at previous time steps are framed as input features to the model. This is the classical fixed-window-based approach of inputting sequence prediction problems used by Multilayer Perceptrons. Instead, the sequence should be presented to the model as multiple input time steps (e.g. a many-to-one model). This confusion may lead you to think you have implemented a many-to-one or many-to-many sequence prediction model when in fact you only have a single vector input for one time step.

5.5.2 Time steps as Output Features

Predictions at multiple future time steps are framed as output features to the model. This is the classical fixed-window approach of making multi-step predictions used by Multilayer Perceptrons and other machine learning algorithms. Instead, the sequence predictions should be generated one time step at a time by the model. This confusion may lead you to think you have implemented a one-to-many or many-to-many sequence prediction model when in fact you only have a single vector output for one time step (e.g. seq2vec not seq2seq).

Note: framing time steps as features in sequence prediction problems is a valid strategy, and could lead to improved performance even when using recurrent neural networks (try it!). The important point here is to understand the common pitfalls and not trick yourself when framing your own prediction problems.

5.6 Further Reading

This section provides some resources for further reading.

5.6.1 Articles

- *The Unreasonable Effectiveness of Recurrent Neural Networks*, 2015.
  http://karpathy.github.io/2015/05/21/rnn-effectiveness/

5.7 Extensions

Are you looking to deepen your understanding of the sequence prediction models? This section lists some challenging extensions that you may wish to consider.

- Summarize each of the 4 models as a cheat sheet for yourself.
- List all the ways a multivariate time series could be framed and the different prediction models that could be used.
- Pick one model and one application for that model and draw a diagram with example input and output sequences.
- List 2 new examples of sequence prediction problems for each model.
- Find 5 recent papers on LSTMs on arXiv.org; list the title and which sequence prediction model was discussed.

Post your extensions online and share the link with me. I'd love to see what you come up with!

5.8 Summary

In this lesson, you discovered the 4 models for sequence prediction and how to realize them in Keras. Specifically, you learned:

- The 4 models for sequence prediction and how they may be implemented in Keras.
- Examples of sequence prediction problems and which of the 4 models that they map onto.
- The traps that beginners fall into in applying the sequence prediction models and how to avoid them.

In the next lesson, you will discover the Vanilla LSTM architecture that you can use for most sequence prediction problems.

Part III

Models

Chapter 6

How to Develop Vanilla LSTMs

6.0.1 Lesson Goal

The goal of this lesson is to learn how to develop and evaluate vanilla LSTM models. After completing this lesson, you will know:

- The architecture of the Vanilla LSTM for sequence prediction and its general capabilities.
- How to define and implement the echo sequence prediction problem.
- How to develop a Vanilla LSTM to learn and make accurate predictions on the echo sequence prediction problem.

6.0.2 Lesson Overview

This lesson is divided into 7 parts; they are:

1. The Vanilla LSTM.
2. Echo Sequence Prediction Problem.
3. Define and Compile the Model.
4. Fit the Model.
5. Evaluate the Model.
6. Make Predictions With the Model.
7. Complete Example.

Let's get started.

6.1 The Vanilla LSTM

6.1.1 Architecture

A simple LSTM configuration is the Vanilla LSTM. It is named Vanilla in this book to differentiate it from deeper LSTMs and the suite of more elaborate configurations. It is the LSTM architecture defined in the original 1997 LSTM paper and the architecture that will give good results on most small sequence prediction problems. The Vanilla LSTM is defined as:

1. Input layer.
2. Fully connected LSTM hidden layer.
3. Fully connected output layer.

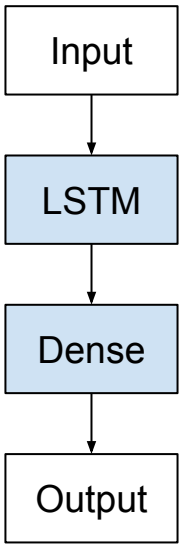

Figure 6.1: Vanilla LSTM Architecture.

6.1.2 Implementation

In Keras, a Vanilla LSTM is defined below, with ellipsis for the specific configuration of the number of neurons in each layer.

```
model = Sequential()
model.add(LSTM(..., input_shape=(...)))
model.add(Dense(...))
```

Listing 6.1: Example of defining a Vanilla LSTM model.

This is the default or standard LSTM referenced in much work and discussion on LSTMs in deep learning and a good starting point when getting started with LSTMs on your sequence prediction problem. The Vanilla LSTM has the following 5 attractive properties, most of which were demonstrated in the original paper:

- Sequence classification conditional on multiple distributed input time steps.
- Memory of precise input observations over thousands of time steps.
- Sequence prediction as a function of prior time steps.
- Robust to the insertion of random time steps on the input sequence.
- Robust to the placement of signal data on the input sequence.

Next, we will define a simple sequence prediction problem that we can later use to demonstrate the Vanilla LSTM.

6.2 Echo Sequence Prediction Problem

The echo sequence prediction problem is a contrived problem for demonstrating the memory capability of the Vanilla LSTM. The task is that, given a sequence of random integers as input, to output the value of a random integer at a specific time input step that is not specified to the model.

For example, given the input sequence of random integers `[5, 3, 2]` and the chosen time step was the second value, then the expected output is 3. Technically, this is a sequence classification problem; it is formulated as a many-to-one prediction problem, where there are multiple input time steps and one output time step at the end of the sequence.

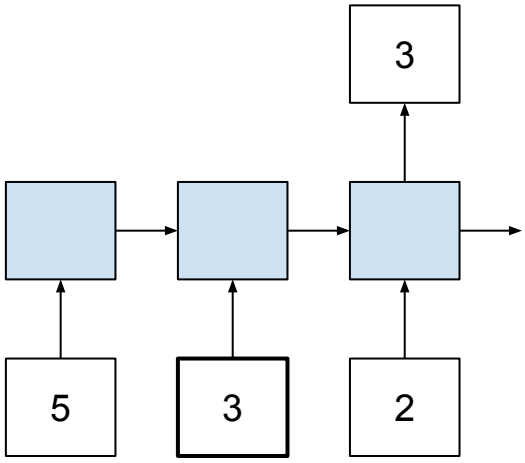

Figure 6.2: Echo sequence prediction problem framed with a many-to-one prediction model.

This problem was carefully chosen to demonstrate the memory capability of the Vanilla LSTM. Further, we will manually perform some of the elements of the model life-cycle such as fitting and evaluating the model to get a deeper feeling for what is happening under the covers. Next, we will develop code to generate examples of this problem. This involves the following steps:

1. Generate Random Sequences.

2. One Hot Encode Sequences.
3. Worked Example
4. Reshape Sequences.

6.2.1 Generate Random Sequences

We can generate random integers in Python using the `randint()` function that takes two parameters indicating the range of integers from which to draw values. In this lesson, we will define the problem as having integer values between 0 and 99 with 100 unique values.

```
randint(0, 99)
```

Listing 6.2: Generate random integers.

We can put this in a function called `generate_sequence()` that will generate a sequence of random integers of the desired length. This function is listed below.

```
# generate a sequence of random numbers in [0, n_features)
def generate_sequence(length, n_features):
  return [randint(0, n_features-1) for _ in range(length)]
```

Listing 6.3: Function to generate sequences of random integers.

6.2.2 One Hot Encode Sequences

Once we have generated sequences of random integers, we need to transform them into a format that is suitable for training an LSTM network. One option would be to rescale the integer to the range `[0,1]`. This would work and would require that the problem be phrased as regression.

We are interested in predicting the right number, not a number close to the expected value. This means we would prefer to frame the problem as classification rather than regression, where the expected output is a class and there are 100 possible class values. In this case, we can use a one hot encoding of the integer values where each value is represented by a 100 element binary vector that is all `0` values except the index of the integer, which is marked 1.

The function below called `one_hot_encode()` defines how to iterate over a sequence of integers and create a binary vector representation for each and returns the result as a 2-dimensional array.

```
# one hot encode sequence
def one_hot_encode(sequence, n_features):
  encoding = list()
  for value in sequence:
    vector = [0 for _ in range(n_features)]
    vector[value] = 1
    encoding.append(vector)
  return array(encoding)
```

Listing 6.4: Function to one hot encode sequences of random integers.

We also need to decode the encoded values so that we can make use of the predictions; in this case, to just review them. The one hot encoding can be inverted by using the `argmax()` NumPy function that returns the index of the value in the vector with the largest value. The

function below, named `one_hot_decode()`, will decode an encoded sequence and can be used to later decode predictions from our network.

```
# decode a one hot encoded string
def one_hot_decode(encoded_seq):
  return [argmax(vector) for vector in encoded_seq]
```

Listing 6.5: Function to decode encoded sequences.

6.2.3 Worked Example

We can tie all of this together. Below is the complete code listing for generating a sequence of 25 random integers and encoding each integer as a binary vector.

```
from random import randint
from numpy import array
from numpy import argmax

# generate a sequence of random numbers in [0, n_features)
def generate_sequence(length, n_features):
  return [randint(0, n_features-1) for _ in range(length)]

# one hot encode sequence
def one_hot_encode(sequence, n_features):
  encoding = list()
  for value in sequence:
    vector = [0 for _ in range(n_features)]
    vector[value] = 1
    encoding.append(vector)
  return array(encoding)

# decode a one hot encoded string
def one_hot_decode(encoded_seq):
  return [argmax(vector) for vector in encoded_seq]

# generate random sequence
sequence = generate_sequence(25, 100)
print(sequence)
# one hot encode
encoded = one_hot_encode(sequence, 100)
print(encoded)
# one hot decode
decoded = one_hot_decode(encoded)
print(decoded)
```

Listing 6.6: Example of generating sequences and encoding them.

Running the example first prints the list of 25 random integers, followed by a truncated view of the binary representations of all integers in the sequence, one vector per line, then the decoded sequence again. You may get different results as different random integers are generated each time the code is run.

```
[37, 99, 40, 98, 44, 27, 99, 18, 52, 97, 46, 39, 60, 13, 66, 29, 26, 4, 65, 85, 29, 88, 8,
    23, 61]
[[0 0 0 ..., 0 0 0]
```

```
[0 0 0 ..., 0 0 1]
[0 0 0 ..., 0 0 0]
...,
[0 0 0 ..., 0 0 0]
[0 0 0 ..., 0 0 0]
[0 0 0 ..., 0 0 0]]
[37, 99, 40, 98, 44, 27, 99, 18, 52, 97, 46, 39, 60, 13, 66, 29, 26, 4, 65, 85, 29, 88, 8,
    23, 61]
```

Listing 6.7: Example output from generating sequences and encoding them.

6.2.4 Reshape Sequences

The final step is to reshape the one hot encoded sequences into a format that can be used as input to the LSTM. This involves reshaping the encoded sequence to have `n` time steps and `k` features, where `n` is the number of integers in the generated sequence and `k` is the set of possible integers at each time step (e.g. 100)

A sequence can then be reshaped into a three-dimensional matrix of samples, time steps, and features, or for a single sequence of 25 integers `[1, 25, 100]`. As follows

```
X = encoded.reshape(1, 25, 100)
```

Listing 6.8: Example of reshaping an encoded sequence.

The output for the sequence is simply the encoded integer at a specific pre-defined location. This location must remain consistent for all examples generated for one model, so that the model can learn. For example, we can use the 2nd time step as the output of a sequence with 25 time steps by taking the encoded value directly from the encoded sequence

```
y = encoded[1, :]
```

Listing 6.9: Example accessing a value in the encoded sequence.

We can put this and the above generation and encoding steps together into a new function called `generate_example()` that generates a sequence, encodes it, and returns the input (`X`) and output (`y`) components for training an LSTM.

```
# generate one example for an lstm
def generate_example(length, n_features, out_index):
  # generate sequence
  sequence = generate_sequence(length, n_features)
  # one hot encode
  encoded = one_hot_encode(sequence, n_features)
  # reshape sequence to be 3D
  X = encoded.reshape((1, length, n_features))
  # select output
  y = encoded[out_index].reshape(1, n_features)
  return X, y
```

Listing 6.10: Function to generate sequences, encode them and reshape them.

We can put all of this together and test the generation of one example ready for fitting or evaluating an LSTM as follows:

```
from random import randint
from numpy import array
from numpy import argmax

# generate a sequence of random numbers in [0, n_features)
def generate_sequence(length, n_features):
  return [randint(0, n_features-1) for _ in range(length)]

# one hot encode sequence
def one_hot_encode(sequence, n_features):
  encoding = list()
  for value in sequence:
    vector = [0 for _ in range(n_features)]
    vector[value] = 1
    encoding.append(vector)
  return array(encoding)

# decode a one hot encoded string
def one_hot_decode(encoded_seq):
  return [argmax(vector) for vector in encoded_seq]

# generate one example for an lstm
def generate_example(length, n_features, out_index):
  # generate sequence
  sequence = generate_sequence(length, n_features)
  # one hot encode
  encoded = one_hot_encode(sequence, n_features)
  # reshape sequence to be 3D
  X = encoded.reshape((1, length, n_features))
  # select output
  y = encoded[out_index].reshape(1, n_features)
  return X, y

X, y = generate_example(25, 100, 2)
print(X.shape)
print(y.shape)
```

Listing 6.11: Example of testing the function to generate encoded sequences and reshape them.

Running the code generates one encoded sequence and prints out the shape of the input and output components of the sequence for the LSTM.

```
(1, 25, 100)
(1, 100)
```

Listing 6.12: Example output from generating encoded sequences and reshaping them.

Now that we know how to prepare and represent random sequences of integers, we can look at using LSTMs to learn them.

6.3 Define and Compile the Model

We will start off by defining and compiling the model. To keep the model small and ensure it is fit in a reasonable time, we will greatly simplify the problem by reducing the sequence length to

5 integers and the number of features to 10 (e.g. 0-9). The model must specify the expected dimensionality of the input data. In this case, in terms of time steps (5) and features (10). We will use a single hidden layer LSTM with 25 memory units, chosen with a little trial and error.

The output layer is a fully connected layer (`Dense`) with 10 neurons for the 10 possible integers that may be output. A `softmax` activation function is used on the output layer to allow the network to learn and output the distribution over the possible output values.

The network will use the log loss function while training, suitable for multiclass classification problems, and the efficient Adam optimization algorithm. The accuracy metric will be reported each training epoch to give an idea of the skill of the model in addition to the loss.

```
# define model
length = 5
n_features = 10
out_index = 2
model = Sequential()
model.add(LSTM(25, input_shape=(length, n_features)))
model.add(Dense(n_features, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Listing 6.13: Example of defining a Vanilla LSTM for the Echo Problem.

Running the example defines and compiles the model, then prints a summary of the model structure. Printing a summary of the model structure is a good practice in general to confirm the model was defined and compiled as you intended.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 25)                3600
_________________________________________________________________
dense_1 (Dense)              (None, 10)                260
=================================================================
Total params: 3,860
Trainable params: 3,860
Non-trainable params: 0
_________________________________________________________________
```

Listing 6.14: Example output from the defined model.

6.4 Fit the Model

We can now fit the model on example sequences. The code we developed for the echo sequence prediction problem generates random sequences. We could generate a large number of example sequences and pass them to the model's `fit()` function. The dataset would be loaded into memory, training would be fast, and we could experiment with varied number of epochs vs dataset size and number of batches.

A simpler approach is to manage the training process manually where one training sample is generated and used to update the model and any internal state is cleared. The number of epochs is the number of iterations of generating samples and essentially the batch size is 1 sample. Below is an example of fitting the model for 10,000 epochs found with a little trial and error.

```
# fit model
for i in range(10000):
  X, y = generate_example(length, n_features, out_index)
  model.fit(X, y, epochs=1, verbose=2)
```

Listing 6.15: Example of fitting the defined LSTM model.

Fitting the model will report the log loss and accuracy for each pattern. Here, accuracy is either 0 or 1 (0% or 100%) because we are making sequence classification prediction on one sample and reporting the result.

```
...
Epoch 1/1
0s - loss: 0.1610 - acc: 1.0000
Epoch 1/1
0s - loss: 0.0288 - acc: 1.0000
Epoch 1/1
0s - loss: 0.0166 - acc: 1.0000
Epoch 1/1
0s - loss: 0.0013 - acc: 1.0000
Epoch 1/1
0s - loss: 0.0244 - acc: 1.0000
```

Listing 6.16: Example output from the fitting the defined model.

6.5 Evaluate the Model

Once the model is fit, we can estimate the skill of the model when classifying new random sequences. We can do this by simply making predictions on 100 randomly generated sequences and counting the number of correct predictions made.

As with fitting the model, we could generate a large number of examples, concatenate them together, and use the `evaluate()` function to evaluate the model. In this case, we will make the predictions manually and count up the number of correct outcomes. We can do this in a loop that generates a sample, makes a prediction, and increments a counter if the prediction was correct.

```
# evaluate model
correct = 0
for i in range(100):
  X, y = generate_example(length, n_features, out_index)
  yhat = model.predict(X)
  if one_hot_decode(yhat) == one_hot_decode(y):
    correct += 1
print('Accuracy: %f' % ((correct/100)*100.0))
```

Listing 6.17: Example of evaluating the fit LSTM model.

Evaluating the model reports the estimated skill of the model as 100%.

```
Accuracy: 100.000000
```

Listing 6.18: Example output from evaluating the fit model.

6.6 Make Predictions With the Model

Finally, we can use the fit model to make predictions on new randomly generated sequences. For this problem, this is much the same as the case of evaluating the model. Because this is more of a user-facing activity, we can decode the whole sequence, expected output, and prediction and print them on the screen.

```
# prediction on new data
X, y = generate_example(length, n_features, out_index)
yhat = model.predict(X)
print('Sequence: %s' % [one_hot_decode(x) for x in X])
print('Expected: %s' % one_hot_decode(y))
print('Predicted: %s' % one_hot_decode(yhat))
```

Listing 6.19: Example of making predictions with the fit LSTM model.

Running the example will print the decoded randomly generated sequence, expected outcome, and (hopefully) a prediction that meets the expected value. Your specific results will vary.

```
Sequence: [[7, 0, 2, 6, 7]]
Expected: [2]
Predicted: [2]
```

Listing 6.20: Example output from making predictions the fit model.

Don't panic if the model gets it wrong. LSTMs are stochastic and it is possible that a single run of the model may converge on a solution that does not completely learn the problem. If this happens to you, try running the example a few more times.

6.7 Complete Example

This section lists the complete working example for your reference.

```
from random import randint
from numpy import array
from numpy import argmax
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# generate a sequence of random numbers in [0, n_features)
def generate_sequence(length, n_features):
  return [randint(0, n_features-1) for _ in range(length)]

# one hot encode sequence
def one_hot_encode(sequence, n_features):
  encoding = list()
  for value in sequence:
    vector = [0 for _ in range(n_features)]
    vector[value] = 1
    encoding.append(vector)
  return array(encoding)

# decode a one hot encoded string
def one_hot_decode(encoded_seq):
```

```
  return [argmax(vector) for vector in encoded_seq]

# generate one example for an lstm
def generate_example(length, n_features, out_index):
  # generate sequence
  sequence = generate_sequence(length, n_features)
  # one hot encode
  encoded = one_hot_encode(sequence, n_features)
  # reshape sequence to be 3D
  X = encoded.reshape((1, length, n_features))
  # select output
  y = encoded[out_index].reshape(1, n_features)
  return X, y

# define model
length = 5
n_features = 10
out_index = 2
model = Sequential()
model.add(LSTM(25, input_shape=(length, n_features)))
model.add(Dense(n_features, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()

# fit model
for i in range(10000):
  X, y = generate_example(length, n_features, out_index)
  model.fit(X, y, epochs=1, verbose=2)

# evaluate model
correct = 0
for i in range(100):
  X, y = generate_example(length, n_features, out_index)
  yhat = model.predict(X)
  if one_hot_decode(yhat) == one_hot_decode(y):
    correct += 1
print('Accuracy: %f' % ((correct/100.0)*100.0))

# prediction on new data
X, y = generate_example(length, n_features, out_index)
yhat = model.predict(X)
print('Sequence: %s' % [one_hot_decode(x) for x in X])
print('Expected: %s' % one_hot_decode(y))
print('Predicted: %s' % one_hot_decode(yhat))
```

Listing 6.21: Example of the Vanilla LSTM applied to the Echo Problem.

6.8 Further Reading

This section provides some resources for further reading.

- *Long Short-Term Memory*, 1997.
- *Learning to Forget: Continual Prediction with LSTM*, 1999.

6.9 Extensions

Do you want to dive deeper into the Vanilla LSTM? This section lists some challenging extensions to this lesson.

- Update the example to use a longer sequence length and still achieve 100% accuracy.
- Update the example to use a larger number of features and still achieve 100% accuracy.
- Update the example to use the SGD optimization algorithm and tune the learning rate and momentum.
- Update the example to prepare a large dataset of examples to fit the model and explore different batch sizes.
- Vary the time step index of the sequence output and training epochs to see if there is a relationship between the index and how hard the problem is to learn.

Post your extensions online and share the link with me. I'd love to see what you come up with!

6.10 Summary

In this lesson, you discovered how to develop a Vanilla or standard LSTM. Specifically, you learned:

- The architecture of the Vanilla LSTM for sequence prediction and its general capabilities.
- How to define and implement the echo sequence prediction problem.
- How to develop a Vanilla LSTM to learn and make accurate predictions on the echo sequence prediction problem.

In the next lesson, you will discover how to develop and evaluate the Stacked LSTM model.

Chapter 7

How to Develop Stacked LSTMs

7.0.1 Lesson Goal

The goal of this lesson is to learn how to develop and evaluate stacked LSTM models. After completing this lesson, you will know:

- The motivation for creating a multilayer LSTM and how to develop Stacked LSTM models in Keras.
- The damped sine wave prediction problem and how to prepare examples for fitting LSTM models.
- How to develop, fit, and evaluate a Stacked LSTM model for the damped sine wave prediction problem.

7.0.2 Lesson Overview

This lesson is divided into 7 parts; they are:

1. The Stacked LSTM.
2. Damped Sine Wave Prediction Problem.
3. Define and Compile the Model.
4. Fit the Model.
5. Evaluate the Model.
6. Make Predictions With the Model.
7. Complete Example.

Let's get started.

7.1 The Stacked LSTM

The Stacked LSTM is a model that has multiple hidden LSTM layers where each layer contains multiple memory cells. We will refer to it as a Stacked LSTM here to differentiate it from the unstacked LSTM (Vanilla LSTM) and a variety of other extensions to the basic LSTM model.

7.1.1 Why Increase Depth?

Stacking LSTM hidden layers makes the model deeper, more accurately earning the description as a deep learning technique. It is the depth of neural networks that is generally attributed to the success of the approach on a wide range of challenging prediction problems.

> [the success of deep neural networks] is commonly attributed to the hierarchy that is introduced due to the several layers. Each layer processes some part of the task we wish to solve, and passes it on to the next. In this sense, the DNN can be seen as a processing pipeline, in which each layer solves a part of the task before passing it on to the next, until finally the last layer provides the output.
>
> — *Training and Analyzing Deep Recurrent Neural Networks*, 2013

Additional hidden layers can be added to a Multilayer Perceptron neural network to make it deeper. The additional hidden layers are understood to recombine the learned representation from prior layers and create new representations at high levels of abstraction. For example, from lines to shapes to objects.

A sufficiently large single hidden layer Multilayer Perceptron can be used to approximate most functions. Increasing the depth of the network provides an alternate solution that requires fewer neurons and trains faster. Ultimately, adding depth it is a type of representational optimization.

> Deep learning is built around a hypothesis that a deep, hierarchical model can be exponentially more efficient at representing some functions than a shallow one.
>
> — *How to Construct Deep Recurrent Neural Networks*, 2013

7.1.2 Architecture

The same benefits can be harnessed with LSTMs. Given that LSTMs operate on sequence data, it means that the addition of layers adds levels of abstraction of input observations over time. In effect, chunking observations over time or representing the problem at different time scales.

> ... building a deep RNN by stacking multiple recurrent hidden states on top of each other. This approach potentially allows the hidden state at each level to operate at different timescale
>
> — *How to Construct Deep Recurrent Neural Networks*, 2013

Stacked LSTMs or Deep LSTMs were introduced by Graves, et al. in their application of LSTMs to speech recognition, beating a benchmark on a challenging standard problem.

> RNNs are inherently deep in time, since their hidden state is a function of all previous hidden states. The question that inspired this paper was whether RNNs could also benefit from depth in space; that is from stacking multiple recurrent hidden layers on top of each other, just as feedforward layers are stacked in conventional deep networks.
>
> — *Speech Recognition With Deep Recurrent Neural Networks*, 2013

In the same work, they found that the depth of the network was more important than the number of memory cells in a given layer to model skill. Stacked LSTMs are now a stable technique for challenging sequence prediction problems. A Stacked LSTM architecture can be defined as an LSTM model comprised of multiple LSTM layers. An LSTM layer above provides a sequence output rather than a single value output to the LSTM layer below. Specifically, one output per input time step, rather than one output time step for all input time steps.

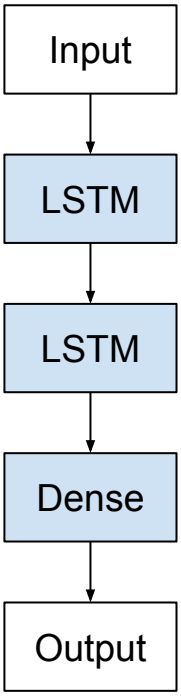

Figure 7.1: Stacked LSTM Architecture.

7.1.3 Implementation

We can easily create Stacked LSTM models in Keras. Each LSTMs memory cell requires a 3D input. When an LSTM processes one input sequence of time steps, each memory cell will output a single value for the whole sequence as a 2D array. We can demonstrate this below with a model that has a single hidden `LSTM` layer that is also the output layer.

```
# Example of one output for whole sequence
from keras.models import Sequential
from keras.layers import LSTM
from numpy import array
```

```
# define model where LSTM is also output layer
model = Sequential()
model.add(LSTM(1, input_shape=(3,1)))
model.compile(optimizer='adam', loss='mse')
# input time steps
data = array([0.1, 0.2, 0.3]).reshape((1,3,1))
# make and show prediction
print(model.predict(data))
```

Listing 7.1: Example of a single layer LSTM.

The input sequence has 3 values. Running the example outputs a single value for the input sequence as a 2D array.

```
[[ 0.00031043]]
```

Listing 7.2: Example output from a single layer LSTM model.

To stack LSTM layers, we need to change the configuration of the prior `LSTM` layer to output a 3D array as input for the subsequent layer. We can do this by setting the `return_sequences` argument on the layer to `True` (defaults to `False`). This will return one output for each input time step and provide a 3D array. Below is the same example as above with `return_sequences=True`.

```
# Example of one output for each input time step
from keras.models import Sequential
from keras.layers import LSTM
from numpy import array
# define model where LSTM is also output layer
model = Sequential()
model.add(LSTM(1, return_sequences=True, input_shape=(3,1)))
model.compile(optimizer='adam', loss='mse')
# input time steps
data = array([0.1, 0.2, 0.3]).reshape((1,3,1))
# make and show prediction
print(model.predict(data))
```

Listing 7.3: Example of an layer LSTM that returns sequences.

Running the example outputs a single value for each time step in the input sequence.

```
[[[-0.02115841]
  [-0.05322712]
  [-0.08976141]]]
```

Listing 7.4: Example output from an LSTM model that returns sequences.

Below is an example of defining a two hidden layer Stacked LSTM:

```
model = Sequential()
model.add(LSTM(..., return_sequences=True, input_shape=(...)))
model.add(LSTM(...))
model.add(Dense(...))
```

Listing 7.5: Example of defining a Stacked LSTM with 2 hidden layers.

We can continue to add hidden LSTM layers as long as the prior `LSTM` layer provides a 3D output as input for the subsequent layer; for example, below is a Stacked LSTM with 4 hidden layers.

```
model = Sequential()
model.add(LSTM(..., return_sequences=True, input_shape=(...)))
model.add(LSTM(..., return_sequences=True))
model.add(LSTM(..., return_sequences=True))
model.add(LSTM(...))
model.add(Dense(...))
```

Listing 7.6: Example of defining a Stacked LSTM with 4 hidden layers.

Next we will define a problem on which we can demonstrate the Stacked LSTM.

7.2 Damped Sine Wave Prediction Problem

This section describes and implements the damped sine wave prediction problem. This section is divided into the following parts:

1. Sine Wave.
2. Damped Sine Wave.
3. Random Damped Sine Waves.
4. Sequences of Damped Sine Waves.

7.2.1 Sine Wave

A sine wave describes an oscillation over time that has a consistent amplitude (movement from baseline) and frequency (time steps between minimum and maximum values). Without getting bogged down in the equation of a sine wave, we can prepare code to create a sine wave as a sequence and plot it.

```
from math import sin
from math import pi
from matplotlib import pyplot
# create sequence
length = 100
freq = 5
sequence = [sin(2 * pi * freq * (i/float(length))) for i in range(length)]
# plot sequence
pyplot.plot(sequence)
pyplot.show()
```

Listing 7.7: Example of generating and plotting a sine wave.

Running the example creates a plot of a sine wave.

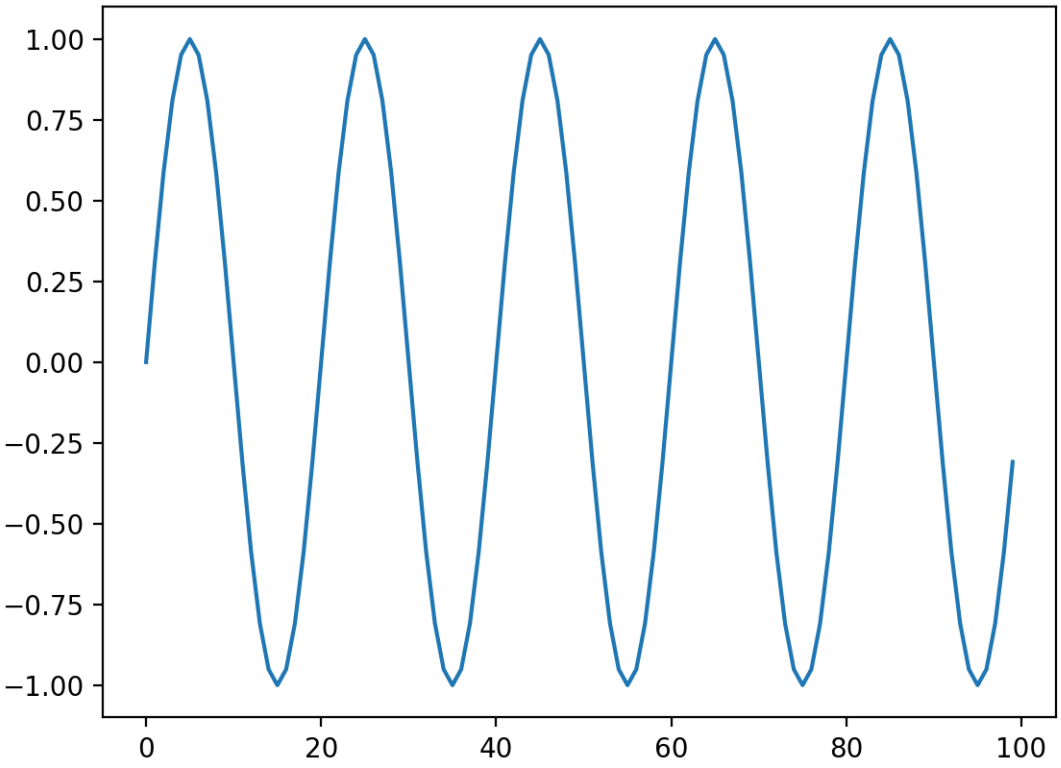

Figure 7.2: Line plot of a generated sine wave.

We can see that the sine wave sequence has the properties that it varies over time with upward and downward movement. This is good local movement that a Vanilla LSTM can model. An LSTM could memorize the sequence or it can use the last few time steps to predict the next time step.

7.2.2 Damped Sine Wave

There is a type of sine wave that decreases with time. The decrease in amplitude provides an additional longer term movement that may require an additional level of abstraction in the LSTM to learn. Again, without going into the equation, we can implement this in Python as follows. The implementation was careful to ensure all values are in the range between 0 and 1.

```
from math import sin
from math import pi
from math import exp
from matplotlib import pyplot
# create sequence
length = 100
period = 10
decay = 0.05
sequence = [0.5 + 0.5 * sin(2 * pi * i / period) * exp(-decay * i) for i in range(length)]
# plot sequence
```

```
pyplot.plot(sequence)
pyplot.show()
```

Listing 7.8: Example of generating and plotting a damped sine wave.

Running the example shows the sine wave and the dampening effect over time. The period parameter determines how many time steps before one whole cycle completes and the decay parameter determines how quickly the series decreases toward zero.

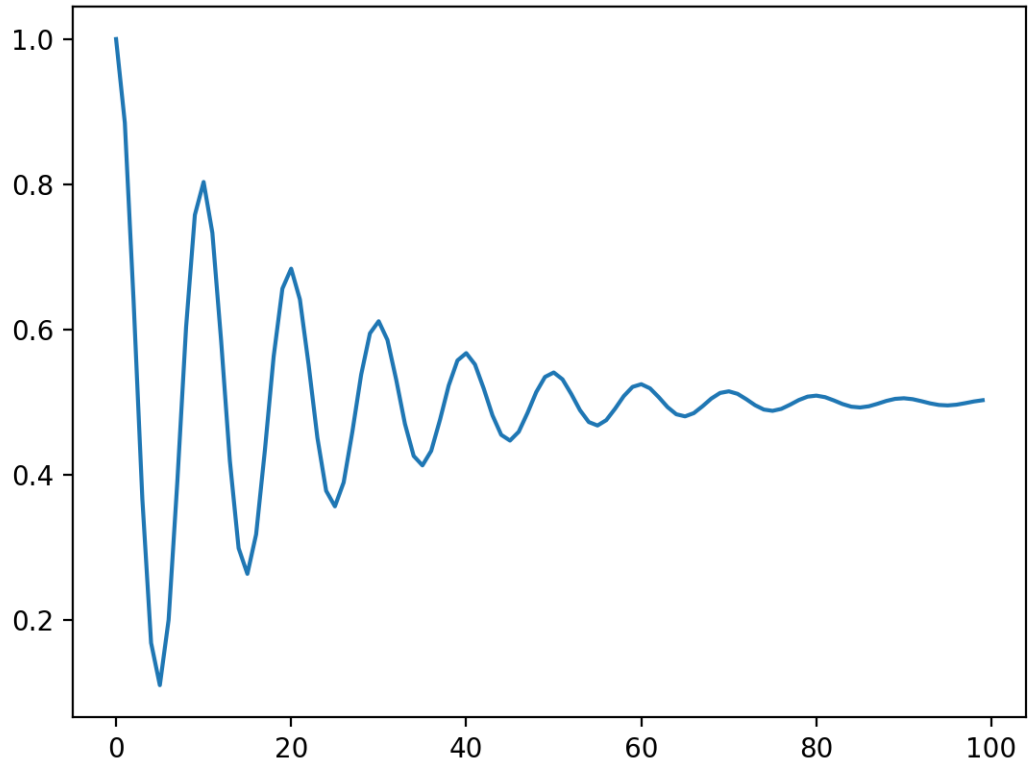

Figure 7.3: Line plot of a generated damped sine wave.

We will use this as a sequence prediction problem to demonstrate Stacked LSTMs.

7.2.3 Random Damped Sine Waves

We can generate sequences of damped sine waves with different periods and decay values, withhold the last few points of the series, and have the model predict them. First, we can define a function to generate a damped sine wave called `generate_sequence()`.

```
# generate damped sine wave in [0,1]
def generate_sequence(length, period, decay):
  return [0.5 + 0.5 * sin(2 * pi * i / period) * exp(-decay * i) for i in range(length)]
```

Listing 7.9: Function for generating a parameterized damped sine wave.

7.2.4 Sequences of Damped Sine Waves

Next, we need a function to generate sequences with a randomly selected period and decay. We will select a uniformly random period between 10 and 20 using the `randint()` function and a uniformly random decay between. 0.01 and 0.1 using the `uniform()` function.

```
p = randint(10, 20)
d = uniform(0.01, 0.1)
```

Listing 7.10: Example of generating random parameters for the damped sign wave.

We will split out the last `n` time steps of each sequence and use that as the output sequence to be predicted. We will make this configurable along with the number of sequences to generate and the length of the sequences. The function below named `generate_examples()` implements this and returns an array of input and output examples ready for training or evaluating an LSTM.

```
# generate input and output pairs of damped sine waves
def generate_examples(length, n_patterns, output):
  X, y = list(), list()
  for _ in range(n_patterns):
    p = randint(10, 20)
    d = uniform(0.01, 0.1)
    sequence = generate_sequence(length + output, p, d)
    X.append(sequence[:-output])
    y.append(sequence[-output:])
  X = array(X).reshape(n_patterns, length, 1)
  y = array(y).reshape(n_patterns, output)
  return X, y
```

Listing 7.11: Example of a function for generating random examples of damped sign wave sequences.

We can test this function by generating a few examples and plotting the sequences.

```
from math import sin
from math import pi
from math import exp
from random import randint
from random import uniform
from numpy import array
from matplotlib import pyplot

# generate damped sine wave in [0,1]
def generate_sequence(length, period, decay):
  return [0.5 + 0.5 * sin(2 * pi * i / period) * exp(-decay * i) for i in range(length)]

# generate input and output pairs of damped sine waves
def generate_examples(length, n_patterns, output):
  X, y = list(), list()
  for _ in range(n_patterns):
    p = randint(10, 20)
    d = uniform(0.01, 0.1)
    sequence = generate_sequence(length + output, p, d)
    X.append(sequence[:-output])
    y.append(sequence[-output:])
  X = array(X).reshape(n_patterns, length, 1)
```

```
  y = array(y).reshape(n_patterns, output)
  return X, y

# test problem generation
X, y = generate_examples(20, 5, 5)
for i in range(len(X)):
  pyplot.plot([x for x in X[i, :, 0]] + [x for x in y[i]], '-o')
pyplot.show()
```

Listing 7.12: Example of generating and plotting a multiple random damped sine wave sequences.

Running the example creates 5 damped sine wave sequences each with 20 time steps. An additional 5 time steps are generated at the end of the sequence that will be held back as test data. We will scale this to 50 time steps for the actual problem.

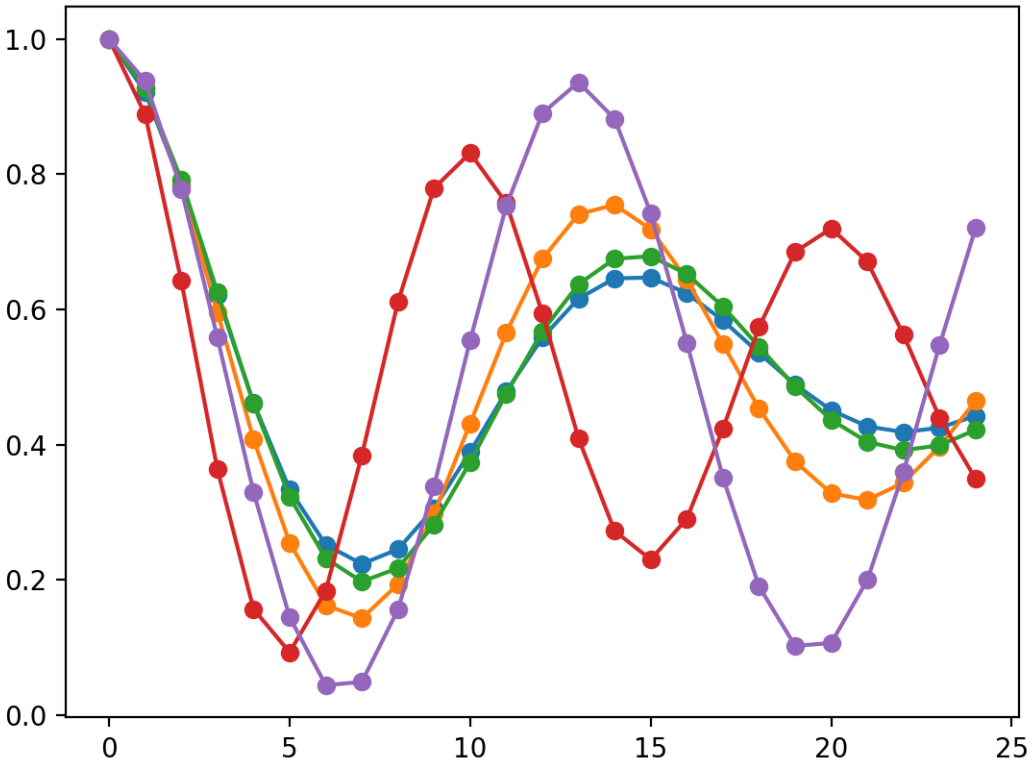

Figure 7.4: Line plot of multiple randomly generated damped sine waves, with dots showing sequence values.

This is a regression type sequence prediction problem. It may also be considered a time series forecasting regression problem. In time series forecasting, it is good practice to make the series stationary, that is remove any systematic trends and seasonality from the series before modeling the problem. This is recommended when working with LSTMs. We are intentionally not making the series stationary in this lesson to demonstrate the capability of the Stacked LSTM.

Technically this is a many-to-one sequence prediction problem. This may be confusing because we clearly intend to predict a sequence of output time steps. The reason that this is a many-to-one prediction problem is because the model will not predict the output time steps piecewise; the whole prediction will be produced at once.

From a model perspective, a sequence of `n` time steps is fed in, then at the end of the sequence a single prediction is made; it just so happens that the prediction is a vector of `n` features that we will interpret as time steps. We could adapt the model to be many-to-many with architectural changes to the proposed LSTM. Consider exploring this as an extension.

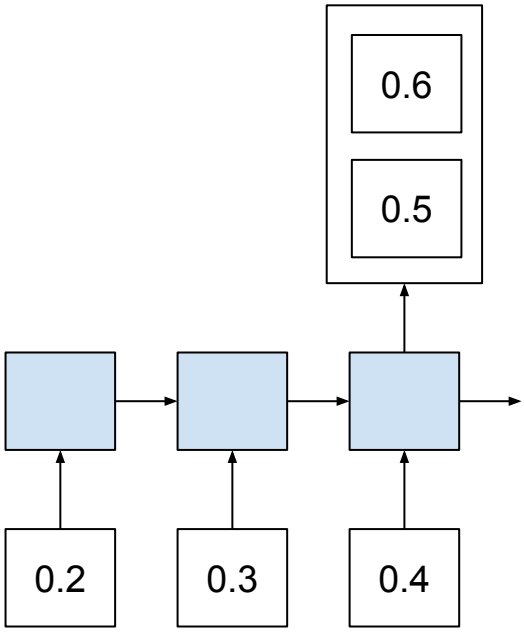

Figure 7.5: Damped sign wave prediction problem framed with a many-to-one prediction model.

Next we can develop a Stacked LSTM model for this problem.

7.3 Define and Compile the Model

We will define a Stacked LSTM with two hidden `LSTM` layers. Each `LSTM` layer will have 20 memory cells. The input dimension will be 1 feature with 20 time steps. The output dimension of the model will be a vector of 5 values that we will interpret to be 5 time steps. The output layer will use the linear activation function, which is the default used when no function is specified.

The Mean Absolute Error (`mae`) loss function will be optimized and the Adam implementation of the gradient descent optimization algorithm will be used. The code for the model definition is listed below.

```
# configure problem
length = 50
output = 5

# define model
```

```
model = Sequential()
model.add(LSTM(20, return_sequences=True, input_shape=(length, 1)))
model.add(LSTM(20))
model.add(Dense(output))
model.compile(loss='mae', optimizer='adam')
model.summary()
```

Listing 7.13: Define a Stacked LSTM model for the damped sine wave problem.

This model configuration, specifically the use of 2 layers and the use of 20 cells per layer, was chosen after a little trial and error. The model configuration is competent, but by no means tuned for this problem. Running just this portion defines and compiles the model, then prints a summary of the model structure. In the structure, we can confirm the shape of the inputs and outputs of the model and the use of two hidden `LSTM` layers.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 50, 20)            1760
_________________________________________________________________
lstm_2 (LSTM)                (None, 20)                3280
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 105
=================================================================
Total params: 5,145
Trainable params: 5,145
Non-trainable params: 0
_________________________________________________________________
```

Listing 7.14: Example output defining a Stacked LSTM model.

7.4 Fit the Model

We can now fit the model on a dataset of randomly generated examples of damped sine waves. The model is expected to generalize a solution to predicting the last few time steps of a damped sine wave time series. We could generate a small number of examples and fit the model on those random examples over many epochs. The downside is that model will see the same random examples many times and may try to memorize them.

Alternately, we could generate a large number of random examples and fit the model on one epoch of this dataset. It would require more memory, but may offer faster training and a more generalized solution. This is the approach we will use.

We will generate 10,000 random damped sine wave examples to fit the model and fit the model using one epoch of this dataset. This is like fitting the model for 10,000 epochs. Ideally, we would reset the internal state of the model after each sample by setting the `batch_size` to 1. In this case, we will trade-off purity for training speed and set the `batch_size` to 10. This will mean that the model weights will be updated and the LSTM memory cell internal state will be reset after each 10 samples.

```
# fit model
X, y = generate_examples(length, 10000, output)
model.fit(X, y, batch_size=10, epochs=1)
```

Listing 7.15: Example of fitting the defined Stacked LSTM model.

Training may take a few minutes, and for this reason the progress bar will be shown as the model is fit. If this causes an issue in your notebook or development environment, you can turn it off by setting `verbose=0` in the call to the `fit()` function.

```
10000/10000 [==============================] - 169s - loss: 0.0481
```

Listing 7.16: Example output from fitting the Stacked LSTM model.

7.5 Evaluate the Model

Once the model is fit, we can evaluate it. Here, we generate a new set of 1,000 random sequences and report the Mean Absolute Error (MAE).

```
# evaluate model
X, y = generate_examples(length, 1000, output)
loss = model.evaluate(X, y, verbose=0)
print('MAE: %f' % loss)
```

Listing 7.17: Example of evaluating the fit Stacked LSTM model.

Evaluating the model reports the skill at around 0.02 MAE. The specific skill score may vary when you run it because of the stochastic nature of neural networks.

```
MAE: 0.021665
```

Listing 7.18: Example output from evaluating the Stacked LSTM model.

This is good for comparing models and model configurations by their skill, but it is hard to get an idea of what is going on.

7.6 Make Predictions with the Model

We can get a better idea of how skillful the model is by generating a standalone prediction and plotting it against the expected output sequence. We can call the `generate_examples()` function and generate one example then make a prediction using the fit model. The prediction and expected sequence are then plotted for comparison.

```
# prediction on new data
X, y = generate_examples(length, 1, output)
yhat = model.predict(X, verbose=0)
pyplot.plot(y[0], label='y')
pyplot.plot(yhat[0], label='yhat')
pyplot.legend()
pyplot.show()
```

Listing 7.19: Example of making predictions with the fit Stacked LSTM model and plotting the results.

Generating the plot shows, at least for this run and on this specific example, the prediction appears to be a reasonable fit for the expected sequence.

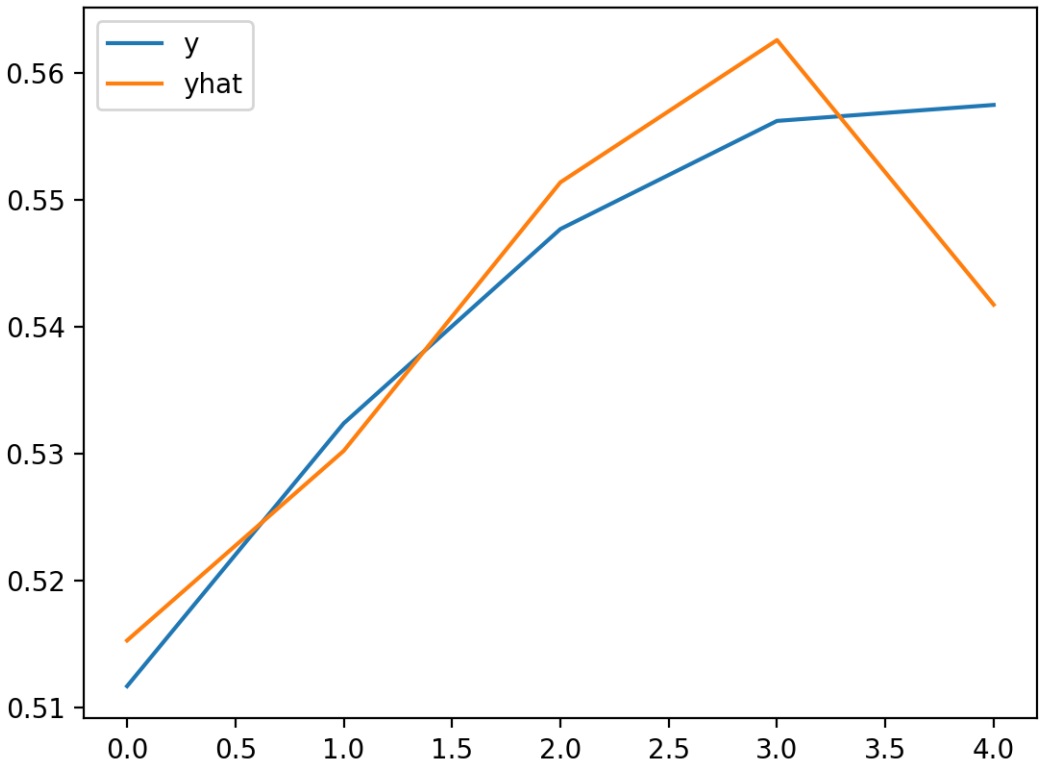

Figure 7.6: Line plot of expected values vs predicted values.

7.7 Complete Example

The complete code example is listed below for your reference.

```
from math import sin
from math import pi
from math import exp
from random import randint
from random import uniform
from numpy import array
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# generate damped sine wave in [0,1]
def generate_sequence(length, period, decay):
  return [0.5 + 0.5 * sin(2 * pi * i / period) * exp(-decay * i) for i in range(length)]
```

```
# generate input and output pairs of damped sine waves
def generate_examples(length, n_patterns, output):
  X, y = list(), list()
  for _ in range(n_patterns):
    p = randint(10, 20)
    d = uniform(0.01, 0.1)
    sequence = generate_sequence(length + output, p, d)
    X.append(sequence[:-output])
    y.append(sequence[-output:])
  X = array(X).reshape(n_patterns, length, 1)
  y = array(y).reshape(n_patterns, output)
  return X, y

# configure problem
length = 50
output = 5

# define model
model = Sequential()
model.add(LSTM(20, return_sequences=True, input_shape=(length, 1)))
model.add(LSTM(20))
model.add(Dense(output))
model.compile(loss='mae', optimizer='adam')
model.summary()

# fit model
X, y = generate_examples(length, 10000, output)
history = model.fit(X, y, batch_size=10, epochs=1)

# evaluate model
X, y = generate_examples(length, 1000, output)
loss = model.evaluate(X, y, verbose=0)
print('MAE: %f' % loss)

# prediction on new data
X, y = generate_examples(length, 1, output)
yhat = model.predict(X, verbose=0)
pyplot.plot(y[0], label='y')
pyplot.plot(yhat[0], label='yhat')
pyplot.legend()
pyplot.show()
```

Listing 7.20: Complete working example of the Stacked LSTM model on the damped sign wave problem.

7.8 Further Reading

This section provides some resources for further reading.

7.8.1 Research Papers

- *How to Construct Deep Recurrent Neural Networks*, 2013.
  https://arxiv.org/abs/1312.6026

- *Training and Analyzing Deep Recurrent Neural Networks*, 2013.
- *Speech Recognition With Deep Recurrent Neural Networks*, 2013.
  https://arxiv.org/abs/1303.5778
- *Generating Sequences With Recurrent Neural Networks*, 2014.
  https://arxiv.org/abs/1308.0850

7.8.2 Articles

- Sine Wave on Wikipedia.
  https://en.wikipedia.org/wiki/Sine_wave
- Damped Sine Wave on Wikipedia.
  https://en.wikipedia.org/wiki/Damped_sine_wave

7.9 Extensions

Do you want to dive deeper into the Stacked LSTM? This section lists some challenging extensions to this lesson.

- List 5 examples that you believe might be a good fit for Stacked LSTMs.
- Tune the number of memory cells, batch size, and number of training samples to further lower the model error (e.g. try 50K samples and a batch size of 1).
- Develop a Vanilla LSTM for the problem and compare the performance of the model.
- Design and execute an experiment to tease out the required increase in training and/or memory cells with the increased length of the damped sine wave sequences.
- Design a new contrived sequence prediction problem tailored for Stacked LSTM and design a Stacked LSTM model to address it skillfully.

Post your extensions online and share the link with me; I'd love to see what you come up with!

7.10 Summary

In this lesson, you discovered how to develop a Stacked LSTM. Specifically, you learned:

- The motivation for creating a multilayer LSTM and how to develop Stacked LSTM models in Keras.
- The damped sine wave prediction problem and how to prepare examples for fitting LSTM models.
- How to develop, fit, and evaluate a Stacked LSTM model for the damped sine wave prediction problem.

In the next lesson, you will discover how to develop and evaluate the CNN LSTM model.

Chapter 8

How to Develop CNN LSTMs

8.0.1 Lesson Goal

The goal of this lesson is to learn how to develop LSTM models that use a Convolutional Neural Network on the front end. After completing this lesson, you will know:

- About the origin of the CNN LSTM architecture and the types of problems to which it is suited.
- How to implement the CNN LSTM architecture in Keras.
- How to develop a CNN LSTM for a Moving Square Video Prediction Problem.

8.0.2 Lesson Overview

This lesson is divided into 7 parts; they are:

1. The CNN LSTM.
2. Moving Square Video Prediction Problem.
3. Define and Compile the Model.
4. Fit the Model.
5. Evaluate the Model.
6. Make Predictions With the Model.
7. Complete Example.

Let's get started.

8.1 The CNN LSTM

8.1.1 Architecture

The CNN LSTM architecture involves using Convolutional Neural Network (CNN) layers for feature extraction on input data combined with LSTMs to support sequence prediction. CNN LSTMs were developed for visual time series prediction problems and the application of generating textual descriptions from sequences of images (e.g. videos). Specifically, the problems of:

- **Activity Recognition**: generating a textual description of an activity demonstrated in a sequence of images.
- **Image Description**: generating a textual description of a single image.
- **Video Description**: generating a textual description of a sequence of images.

> [CNN LSTMs are] a class of models that is both spatially and temporally deep, and has the flexibility to be applied to a variety of vision tasks involving sequential inputs and outputs

— *Long-term Recurrent Convolutional Networks for Visual Recognition and Description*, 2015.

This architecture was originally referred to as a Long-term Recurrent Convolutional Network or LRCN model, although we will use the more generic name *CNN LSTM* to refer to LSTMs that use a CNN as a front end in this lesson. This architecture is used for the task of generating textual descriptions of images. Key is the use of a CNN that is pre-trained on a challenging image classification task that is re-purposed as a feature extractor for the caption generating problem.

> ... it is natural to use a CNN as an image "encoder", by first pre-training it for an image classification task and using the last hidden layer as an input to the RNN decoder that generates sentences

— *Show and Tell: A Neural Image Caption Generator*, 2015.

This architecture has also been used on speech recognition and natural language processing problems where CNNs are used as feature extractors for the LSTMs on audio and textual input data. This architecture is appropriate for problems that:

- Have spatial structure in their input such as the 2D structure or pixels in an image or the 1D structure of words in a sentence, paragraph, or document.
- Have a temporal structure in their input such as the order of images in a video or words in text, or require the generation of output with temporal structure such as words in a textual description.

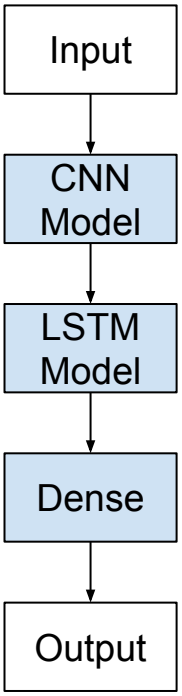

Figure 8.1: CNN LSTM Architecture.

8.1.2 Implementation

We can define a CNN LSTM model to be trained jointly in Keras. A CNN LSTM can be defined by adding CNN layers on the front end followed by `LSTM` layers with a `Dense` layer on the output.

It is helpful to think of this architecture as defining two sub-models: the CNN Model for feature extraction and the LSTM Model for interpreting the features across time steps. Let's take a look at both of these sub models in the context of a sequence of 2D inputs which we will assume are images.

CNN Model

As a refresher, we can define a 2D convolutional network as comprised of `Conv2D` and `MaxPooling2D` layers ordered into a stack of the required depth. The `Conv2D` will interpret snapshots of the image (e.g. small squares) and the pooling layers will consolidate or abstract the interpretation.

For example, the snippet below expects to read in 10×10 pixel images with 1 channel (e.g. black and white). The `Conv2D` will read the image in 2×2 snapshots and output one new 10×10 interpretation of the image. The `MaxPooling2D` will pool the interpretation into 2×2 blocks reducing the output to a 5×5 consolidation. The `Flatten` layer will take the single 5×5 map and transform it into a 25-element vector ready for some other layer to deal with, such as a `Dense` for outputting a prediction.

```
cnn = Sequential()
cnn.add(Conv2D(1, (2,2), activation='relu', padding='same', input_shape=(10,10,1)))
cnn.add(MaxPooling2D(pool_size=(2, 2)))
cnn.add(Flatten())
```

Listing 8.1: Example of the CNN part of the CNN LSTM model.

This makes sense for image classification and other computer vision tasks.

LSTM Model

The CNN model above is only capable of handling a single image, transforming it from input pixels into an internal matrix or vector representation. We need to repeat this operation across multiple images and allow the LSTM to build up internal state and update weights using BPTT across a sequence of the internal vector representations of input images.

The CNN could be fixed in the case of using an existing pre-trained model like VGG for feature extraction from images. The CNN may not be trained, and we may wish to train it by backpropagating error from the LSTM across multiple input images to the CNN model. In both of these cases, conceptually there is a single CNN model and a sequence of LSTM models, one for each time step. We want to apply the CNN model to each input image and pass on the output of each input image to the LSTM as a single time step.

We can achieve this by wrapping the entire CNN input model (one layer or more) in a `TimeDistributed` layer. This layer achieves the desired outcome of applying the same layer or layers multiple times. In this case, applying it multiple times to multiple input time steps and in turn providing a sequence of *image interpretations* or *image features* to the LSTM model to work on.

```
model.add(TimeDistributed(...))
model.add(LSTM(...))
model.add(Dense(...))
```

Listing 8.2: Example of the LSTM part of the CNN LSTM model.

We now have the two elements of the model; let's put them together.

CNN LSTM Model

We can define a CNN LSTM model in Keras by first defining the CNN layer or layers, wrapping them in a `TimeDistributed` layer and then defining the LSTM and output layers. We have two ways to define the model that are equivalent and only differ as a matter of taste. You can define the CNN model first, then add it to the LSTM model by wrapping the entire sequence of CNN layers in a `TimeDistributed` layer, as follows:

```
# define CNN model
cnn = Sequential()
cnn.add(Conv2D(...))
cnn.add(MaxPooling2D(...))
cnn.add(Flatten())
# define LSTM model
model = Sequential()
model.add(TimeDistributed(cnn, ...))
model.add(LSTM(..))
model.add(Dense(...))
```

Listing 8.3: Example of an CNN LSTM model in two parts.

An alternate, and perhaps easier to read, approach is to wrap each layer in the CNN model in a `TimeDistributed` layer when adding it to the main model.

```
model = Sequential()
# define CNN model
model.add(TimeDistributed(Conv2D(...))
model.add(TimeDistributed(MaxPooling2D(...)))
model.add(TimeDistributed(Flatten()))
# define LSTM model
model.add(LSTM(...))
model.add(Dense(...))
```

Listing 8.4: Example of an CNN LSTM model in one part.

The benefit of this second approach is that all of the layers appear in the model summary and as such is preferred for now. You can choose the method that you prefer.

8.2 Moving Square Video Prediction Problem

The moving square video prediction problem is contrived to demonstrate the CNN LSTM. The problem involves the generation of a sequence of frames. In each image a line is drawn from left to right or right to left. Each frame shows the extension of the line by one pixel. The task is for the model to classify whether the line moved left or right in the sequence of frames. Technically, the problem is a sequence classification problem framed with a many-to-one prediction model.

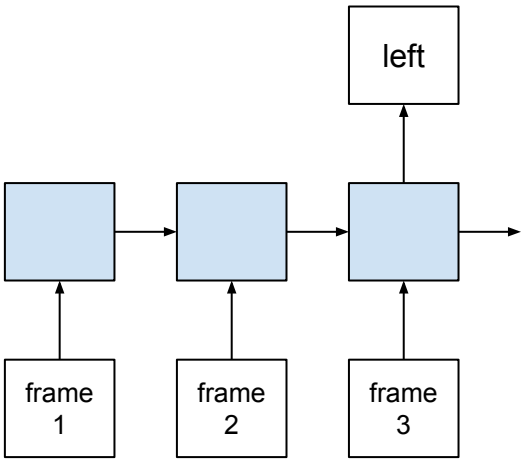

Figure 8.2: Moving square video prediction problem framed with a many-to-one prediction model.

This test problem can be broken down into the following steps:

1. Image Initialization.
2. Adding Steps.
3. Instance Generator.

8.2.1 Image Initialization

We can start off by defining a 2D NumPy array filled with zero values. We will make the images symmetrical, and in this case, 10 pixels by 10 pixels.

```
from numpy import zeros
frame = zeros((10,10))
```

Listing 8.5: Example of creating an empty square image.

Next, we can select the row for the first step of the line. We will use the `randint()` function to select a uniformly random integer between 0 and 9.

```
from random import randint
step = randint(0, 10-1)
```

Listing 8.6: Example of selecting a step.

We can now select whether we are drawing the line left or right across the image. We will use the `random()` function to decide. If right, we will start at the left, or column 0, and if left we will start on the right, or column 9.

```
from random import random
right = 1 if random() < 0.5 else 0
col = 0 if right else size-1
```

Listing 8.7: Example of deciding to move left or right.

We can now mark the start of the line.

```
frame[step, col] = 1
```

Listing 8.8: Example of marking the start of the line.

8.2.2 Adding Steps

Now we need a process to add steps to the line. The next step must be a function of the previous step. We will constrain it to be in the next column along (left or right) and be in the same row, the row above or the row below. We will further constrain the movement by the bounds of the image, e.g. no movements below row zero or above row 9.

We can use the same `randint()` function above to pick the next step and impose our movement constraints on the upper and lower values. The step value chosen last time is stored in the `last_step` variable.

```
lower = max(0, last_step-1)
upper = min(10-1, last_step+1)
step = randint(lower, upper)
```

Listing 8.9: Example of selecting the next step.

Next, we can make a copy of the last image and mark the new position for the next column along.

```
column = i if right else size-1-i
frame = last_frame.copy()
frame[step, column] = 1
```

Listing 8.10: Example of marking the step as a new frame.

This process can be repeated until the first or last column is reached, depending on the chosen direction.

8.2.3 Instance Generator

We can capture all of the above behavior in two small functions. The `build_frames()` function takes an argument to define the size of the images and returns a sequence of images and whether the line moves right (1) or left (0). This function calls another function `next_frame()` to create each subsequent frame after the first frame as the line moves across the image.

To make the problem concrete, we can plot one sequence. We will generate a small sequence with each image 5×5 pixels and with 5 frames and plot the frames side by side.

```
from numpy import zeros
from random import randint
from random import random
from matplotlib import pyplot

# generate the next frame in the sequence
def next_frame(last_step, last_frame, column):
  # define the scope of the next step
  lower = max(0, last_step-1)
  upper = min(last_frame.shape[0]-1, last_step+1)
  # choose the row index for the next step
  step = randint(lower, upper)
  # copy the prior frame
  frame = last_frame.copy()
  # add the new step
  frame[step, column] = 1
  return frame, step

# generate a sequence of frames of a dot moving across an image
def build_frames(size):
  frames = list()
  # create the first frame
  frame = zeros((size,size))
  step = randint(0, size-1)
  # decide if we are heading left or right
  right = 1 if random() < 0.5 else 0
  col = 0 if right else size-1
  frame[step, col] = 1
  frames.append(frame)
  # create all remaining frames
  for i in range(1, size):
    col = i if right else size-1-i
    frame, step = next_frame(step, frame, col)
    frames.append(frame)
  return frames, right

# generate sequence of frames
size = 5
frames, right = build_frames(size)
# plot all frames
pyplot.figure()
for i in range(size):
```

```
  # create a gray scale subplot for each frame
  pyplot.subplot(1, size, i+1)
  pyplot.imshow(frames[i], cmap='Greys')
  # turn of the scale to make it clearer
  ax = pyplot.gca()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
# show the plot
pyplot.show()
```

Listing 8.11: Example of generating a sequence of frames.

Running the example generates a random sequence and plots the frames side by side. You can see that the line wiggles around from left to right across the image, one pixel per time step.

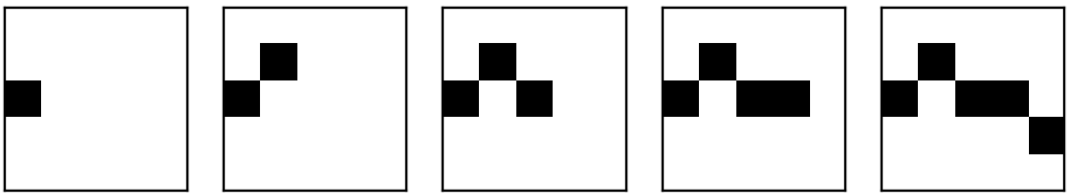

Figure 8.3: Example of a sequence of frames of a line moving across the image.

8.2.4 Prepare Input for Model

Finally, we will prepare a function to generate multiple sequences with the correct shape ready for fitting and evaluating an LSTM model. A function named `generate_examples()` is defined below that takes the size of the images to generate and the number of sequences to generate as arguments.

Each sequence is generated and stored. Importantly, the input sequences for the model must be resized to be suitable for a 2D CNN. Normally this would be:

```
[width, height, channels]
```

Listing 8.12: Example expected input for the 2D CNN model.

In our case, it would be `[size, size, 1]` for the symmetrical black and white images. This is not sufficient as we also have multiple images, then multiple sequences of images. Therefore, the input to the model must be reshaped as:

```
[samples, timesteps, width, height, channels]
```

Listing 8.13: Example of the expected input shape for the model.

Or, in our function:

```
[n_patterns, size, size, size, 1]
```

Listing 8.14: Example of the expected input shape using terms from the problem definition.

This new function for generating random videos is listed below.

```
# generate multiple sequences of frames and reshape for network input
def generate_examples(size, n_patterns):
  X, y = list(), list()
  for _ in range(n_patterns):
    frames, right = build_frames(size)
    X.append(frames)
    y.append(right)
  # resize as [samples, timesteps, width, height, channels]
  X = array(X).reshape(n_patterns, size, size, size, 1)
  y = array(y).reshape(n_patterns, 1)
  return X, y
```

Listing 8.15: Example of generating and reshaping a sequence of frames for the model.

Next we can define and compile the model.

8.3 Define and Compile the Model

We can define a CNN LSTM to fit the model. The size of the generated images controls how challenging the problem will be. We will make the problems modestly challenging by configuring the images to be 50×50 pixels, or a total of 2,500 binary values.

```
# configure problem
size = 50
```

Listing 8.16: Example of configuring the problem.

We will define the model wrapping each layer in the CNN model with a separate `TimeDistributed` layer. This is to ensure that the model summary provides a clear idea of how the network hangs together. We will define a `Conv2D` as an input layer with 2 filters and a 2×2 kernel to pass across the input images. The use of 2 filters was found with some experimentation and it is convention to use small kernel sizes. The `Conv2D` will output 2 49×49 pixel impressions of the input.

Convolutional layers are often immediately followed by a pooling layer. Here we use a `MaxPooling2D` pooling layer with a pool size of 2×2, which will in effect halve the size of each filter output from the previous layer, in turn outputting two 24×24 maps.

The pooling layer is followed by a `Flatten` layer to transform the `[24,24,2]` 3D output from the `MaxPooling2D` layer into a one-dimensional 1,152 element vector. The CNN model is a feature extraction model. The hope is that the vector output of the `Flatten` layer is a compressed and/or more salient representation of the image than the raw pixel values.

Next, we can define the LSTM elements of the model. We will use a single `LSTM` layer with 50 memory cells, configured after a little trial and error. The use of a `TimeDistributed` wrapper around the whole CNN model means that the LSTM will see 50 time steps, with each time step presenting a 1,152 element vector as input.

This is a binary classification problem, so we will use a `Dense` output with a single neuron and the `sigmoid` activation function. The model is compiled to minimize log loss (`binary_crossentropy`) with the Adam implementation of gradient descent and the binary classification accuracy will be reported. The complete code listing is provided below.

```
# define the model
model = Sequential()
model.add(TimeDistributed(Conv2D(2, (2,2), activation='relu'),
    input_shape=(None,size,size,1)))
model.add(TimeDistributed(MaxPooling2D(pool_size=(2, 2))))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(50))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Listing 8.17: Example of defining and compiling the CNN LSTM model.

Running this example prints a summary of the compiled model. We can confirm the expected shape of each layer output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
time_distributed_1 (TimeDist (None, None, 49, 49, 2) 10
_________________________________________________________________
time_distributed_2 (TimeDist (None, None, 24, 24, 2) 0
_________________________________________________________________
time_distributed_3 (TimeDist (None, None, 1152)   0
_________________________________________________________________
lstm_1 (LSTM)                (None, 50)                240600
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 51
=================================================================
Total params: 240,661
Trainable params: 240,661
Non-trainable params: 0
_________________________________________________________________
```

Listing 8.18: Example output from defining the CNN LSTM model.

8.4 Fit the Model

We are now ready to fit the model on randomly generated examples of the problem. The `generate_examples()` function defined above will prepare a specified number of random sequences that we can keep in memory and use to fit the model efficiently. The number of randomly generated examples is a proxy for the number of training epochs as we would prefer the model to be trained on unique problem instances rather than the same set of random instances again and again.

Here we will train the model on a single epoch of 5,000 randomly generated sequences. Ideally, the internal state of the LSTM would be reset at the end of each sequence. We could achieve this by setting the `batch_size` to 1. We will trade off the fidelity of the model for computational efficiency and set the `batch_size` to 32.

```
# fit model
X, y = generate_examples(size, 5000)
model.fit(X, y, batch_size=32, epochs=1)
```

Listing 8.19: Example of fitting the compiled the CNN LSTM model.

Running the example will show a progress bar when executed on the command line indicating the loss and accuracy at the end of each batch. If you are running the example in an IDE or notebook, you can turn off the progress bar by setting `verbose=0`.

```
5000/5000 [==============================] - 37s - loss: 0.1507 - acc: 0.9208
```

Listing 8.20: Example output from fitting the CNN LSTM model.

You could experiment with different numbers of samples, epochs, and batch sizes. Can you develop a skillful model with less overall training?

8.5 Evaluate the Model

Now that the model is fit, we can estimate the skill of the model on new random sequences. Here, we can generate 100 new random sequences and evaluate the accuracy of the model.

```
# evaluate model
X, y = generate_examples(size, 100)
loss, acc = model.evaluate(X, y, verbose=0)
print('loss: %f, acc: %f' % (loss, acc*100))
```

Listing 8.21: Example of evaluating the fit the CNN LSTM model.

Running the example prints both the loss and accuracy of the fit model. Here, we can see the model achieves 100% accuracy. Your results may vary and if you do not see 100% accuracy, try running the example a few times.

```
loss: 0.001120, acc: 100.000000
```

Listing 8.22: Example output from evaluating the CNN LSTM model.

8.6 Make Predictions With the Model

For completeness, we can use the developed model to make predictions on new sequences. Here we generate a new single random sequence and predict whether the line is moving left or right.

```
# prediction on new data
X, y = generate_examples(size, 1)
yhat = model.predict_classes(X, verbose=0)
expected = "Right" if y[0]==1 else "Left"
predicted = "Right" if yhat[0]==1 else "Left"
print('Expected: %s, Predicted: %s' % (expected, predicted))
```

Listing 8.23: Example of making predictions with the fit the CNN LSTM model.

Running the example prints the decoded expected and predicted values.

```
Expected: Right, Predicted: Right
```

Listing 8.24: Example output from making predictions with the fit CNN LSTM model.

8.7 Complete Example

For completeness, the full code listing is provided below for your reference.

```
from random import random
from random import randint
from numpy import array
from numpy import zeros
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import TimeDistributed

# generate the next frame in the sequence
def next_frame(last_step, last_frame, column):
  # define the scope of the next step
  lower = max(0, last_step-1)
  upper = min(last_frame.shape[0]-1, last_step+1)
  # choose the row index for the next step
  step = randint(lower, upper)
  # copy the prior frame
  frame = last_frame.copy()
  # add the new step
  frame[step, column] = 1
  return frame, step

# generate a sequence of frames of a dot moving across an image
def build_frames(size):
  frames = list()
  # create the first frame
  frame = zeros((size,size))
  step = randint(0, size-1)
  # decide if we are heading left or right
  right = 1 if random() < 0.5 else 0
  col = 0 if right else size-1
  frame[step, col] = 1
  frames.append(frame)
  # create all remaining frames
  for i in range(1, size):
    col = i if right else size-1-i
    frame, step = next_frame(step, frame, col)
    frames.append(frame)
  return frames, right

# generate multiple sequences of frames and reshape for network input
def generate_examples(size, n_patterns):
  X, y = list(), list()
  for _ in range(n_patterns):
    frames, right = build_frames(size)
    X.append(frames)
    y.append(right)
  # resize as [samples, timesteps, width, height, channels]
  X = array(X).reshape(n_patterns, size, size, size, 1)
```

```
  y = array(y).reshape(n_patterns, 1)
  return X, y

# configure problem
size = 50

# define the model
model = Sequential()
model.add(TimeDistributed(Conv2D(2, (2,2), activation='relu'),
    input_shape=(None,size,size,1)))
model.add(TimeDistributed(MaxPooling2D(pool_size=(2, 2))))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(50))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()

# fit model
X, y = generate_examples(size, 5000)
model.fit(X, y, batch_size=32, epochs=1)

# evaluate model
X, y = generate_examples(size, 100)
loss, acc = model.evaluate(X, y, verbose=0)
print('loss: %f, acc: %f' % (loss, acc*100))

# prediction on new data
X, y = generate_examples(size, 1)
yhat = model.predict_classes(X, verbose=0)
expected = "Right" if y[0]==1 else "Left"
predicted = "Right" if yhat[0]==1 else "Left"
print('Expected: %s, Predicted: %s' % (expected, predicted))
```

Listing 8.25: Complete examples of the CNN LSTM model on the Moving Square prediction problem.

8.8 Further Reading

This section provides some resources for further reading.

8.8.1 Papers on CNN LSTM

- *Long-term Recurrent Convolutional Networks for Visual Recognition and Description*, 2015.
  https://arxiv.org/abs/1411.4389
- *Show and Tell: A Neural Image Caption Generator*, 2015.
  https://arxiv.org/abs/1411.4555
- *Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks*, 2015.
- *Character-Aware Neural Language Models*, 2015.
  https://arxiv.org/abs/1508.06615

- *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*, 2015.
  https://arxiv.org/abs/1506.04214

8.8.2 Keras API

- `Conv2D` Keras API.
  https://keras.io/layers/convolutional/#conv2d
- `MaxPooling2D` Keras API.
  https://keras.io/layers/pooling/#maxpooling2d
- `Flatten` Keras API.
  https://keras.io/layers/core/#flatten
- `TimeDistributed` Keras API.
  https://keras.io/layers/wrappers/#timedistributed

8.9 Extensions

Do you want to dive deeper into the CNN LSTM? This section lists some challenging extensions to this lesson.

- List 5 computer vision and 5 natural language processing problems where the CNN LSTM model could be applied.
- Tune the number of CNN layers and number of filters and memory cells in layers to see if you can devise a simpler model that is as skillful.
- Update the problem to support traces that move up-down as well as left-right, then tune the model on the new problem.
- Update the problem so that the model only observes a subset of all frames in the sequence and tune the model accordingly.
- Design and execute an experiment that contrasts the CNN LSTM vs the Vanilla LSTM on the contrived problem.

Post your extensions online and share the link with me. I'd love to see what you come up with!

8.10 Summary

In this lesson, you discovered how to develop a CNN LSTM model. Specifically, you learned:

- About the origin of the CNN LSTM architecture and the types of problems to which it is suited.
- How to implement the CNN LSTM architecture in Keras.

- How to develop a CNN LSTM for a Moving Square Video Prediction Problem.

In the next lesson, you will discover how to develop and evaluate the Encoder-Decoder LSTM model.

Chapter 9

How to Develop Encoder-Decoder LSTMs

9.0.1 Lesson Goal

The goal of this lesson is to learn how to develop encoder-decoder LSTM models. After completing this lesson, you will know:

- The Encoder-Decoder LSTM architecture and how to implement it in Keras.
- The addition sequence-to-sequence prediction problem.
- How to develop an Encoder-Decoder LSTM for the addition sequence-to-sequence prediction problem.

9.1 Lesson Overview

This lesson is divided into 7 parts; they are:

1. The Encoder-Decoder LSTM.
2. Addition Prediction Problem.
3. Define and Compile the Model.
4. Fit the Model.
5. Evaluate the Model.
6. Make Predictions With the Model.
7. Complete Example.

Let's get started.

9.2 The Encoder-Decoder LSTM

9.2.1 Sequence-to-Sequence Prediction Problems

Sequence prediction often involves forecasting the next value in a real valued sequence or outputting a class label for an input sequence. This is often framed as a sequence of one input time step to one output time step (e.g. one-to-one) or multiple input time steps to one output time step (many-to-one) type sequence prediction problem.

There is a more challenging type of sequence prediction problem that takes a sequence as input and requires a sequence prediction as output. These are called sequence-to-sequence prediction problems, or seq2seq for short. One modeling concern that makes these problems challenging is that the length of the input and output sequences may vary. Given that there are multiple input time steps and multiple output time steps, this form of problem is referred to as many-to-many type sequence prediction problem.

9.2.2 Architecture

One approach to seq2seq prediction problems that has proven very effective is called the Encoder-Decoder LSTM. This architecture is comprised of two models: one for reading the input sequence and encoding it into a fixed-length vector, and a second for decoding the fixed-length vector and outputting the predicted sequence. The use of the models in concert gives the architecture its name of Encoder-Decoder LSTM designed specifically for seq2seq problems.

> ... RNN Encoder-Decoder, consists of two recurrent neural networks (RNN) that act as an encoder and a decoder pair. The encoder maps a variable-length source sequence to a fixed-length vector, and the decoder maps the vector representation back to a variable-length target sequence.
>
> — *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.

The Encoder-Decoder LSTM was developed for natural language processing problems where it demonstrated state-of-the-art performance, specifically in the area of text translation called statistical machine translation. The innovation of this architecture is the use of a fixed-sized internal representation in the heart of the model that input sequences are read to and output sequences are read from. For this reason, the method may be referred to as sequence embedding.

In one of the first applications of the architecture to English-to-French translation, the internal representation of the encoded English phrases was visualized. The plots revealed a qualitatively meaningful learned structure of the phrases harnessed for the translation task.

> The proposed RNN Encoder-Decoder naturally generates a continuous-space representation of a phrase. [...] From the visualization, it is clear that the RNN Encoder-Decoder captures both semantic and syntactic structures of the phrases
>
> — *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.

On the task of translation, the model was found to be more effective when the input sequence was reversed. Further, the model was shown to be effective even on very long input sequences.

> We were able to do well on long sentences because we reversed the order of words in the source sentence but not the target sentences in the training and test set. By doing so, we introduced many short term dependencies that made the optimization problem much simpler. ... The simple trick of reversing the words in the source sentence is one of the key technical contributions of this work
>
> — *Sequence to Sequence Learning with Neural Networks*, 2014.

This approach has also been used with image inputs where a Convolutional Neural Network is used as a feature extractor on input images, which is then read by a decoder LSTM.

> ... we propose to follow this elegant recipe, replacing the encoder RNN by a deep convolution neural network (CNN). [...] it is natural to use a CNN as an image "encoder", by first pre-training it for an image classification task and using the last hidden layer as an input to the RNN decoder that generates sentences
>
> — *Show and Tell: A Neural Image Caption Generator*, 2014.

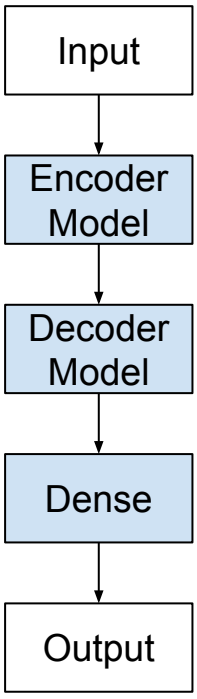

Figure 9.1: Encoder-decoder LSTM Architecture.

9.2.3 Applications

The list below highlights some interesting applications of the Encoder-Decoder LSTM architecture.

- **Machine Translation**, e.g. English to French translation of phrases.

- **Learning to Execute**, e.g. calculate the outcome of small programs.
- **Image Captioning**, e.g. generating a text description for images.
- **Conversational Modeling**, e.g. generating answers to textual questions.
- **Movement Classification**, e.g. generating a sequence of commands from a sequence of gestures.

9.2.4 Implementation

The Encoder-Decoder LSTM can be implemented directly in Keras. We can think of the model as being comprised of two key parts: the encoder and the decoder. First, the input sequence is shown to the network one encoded character at a time. We need an encoding level to learn the relationship between the steps in the input sequence and develop an internal representation of these relationships.

One or more LSTM layers can be used to implement the encoder model. The output of this model is a fixed-size vector that represents the internal representation of the input sequence. The number of memory cells in this layer defines the length of this fixed-sized vector.

```
model = Sequential()
model.add(LSTM(..., input_shape=(...)))
```

Listing 9.1: Example of a Vanilla LSTM model.

The decoder must transform the learned internal representation of the input sequence into the correct output sequence. One or more LSTM layers can also be used to implement the decoder model. This model reads from the fixed sized output from the encoder model. As with the Vanilla LSTM, a `Dense` layer is used as the output for the network. The same weights can be used to output each time step in the output sequence by wrapping the `Dense` layer in a `TimeDistributed` wrapper.

```
model.add(LSTM(..., return_sequences=True))
model.add(TimeDistributed(Dense(...)))
```

Listing 9.2: Example of a LSTM model with `TimeDistributed` wrapped `Dense` layer.

There's a problem though. We must connect the encoder to the decoder, and they do not fit. That is, the encoder will produce a 2-dimensional matrix of outputs, where the length is defined by the number of memory cells in the layer. The decoder is an `LSTM` layer that expects a 3D input of [samples, time steps, features] in order to produce a decoded sequence of some different length defined by the problem.

If you try to force these pieces together, you get an error indicating that the output of the decoder is 2D and 3D input to the decoder is required. We can solve this using a `RepeatVector` layer. This layer simply repeats the provided 2D input multiple times to create a 3D output.

The `RepeatVector` layer can be used like an adapter to fit the encoder and decoder parts of the network together. We can configure the `RepeatVector` to repeat the fixed length vector one time for each time step in the output sequence.

```
model.add(RepeatVector(...))
```

Listing 9.3: Example of a `RepeatVector` layer.

Putting this together, we have:

```
model = Sequential()
model.add(LSTM(..., input_shape=(...)))
model.add(RepeatVector(...))
model.add(LSTM(..., return_sequences=True))
model.add(TimeDistributed(Dense(...)))
```

Listing 9.4: Example of an Encoder-Decoder model.

To summarize, the `RepeatVector` is used as an adapter to fit the fixed-sized 2D output of the encoder to the differing length and 3D input expected by the decoder. The `TimeDistributed` wrapper allows the same output layer to be reused for each element in the output sequence.

9.3 Addition Prediction Problem

The addition problem is a sequence-to-sequence, or seq2seq, prediction problem. It was used by Wojciech Zaremba and Ilya Sutskever in their 2014 paper exploring the capabilities of the Encoder-Decoder LSTM titled "*Learning to Execute*" where the architecture was demonstrated learning to calculate the output of small programs.

The problem is defined as calculating the sum output of two input numbers. This is challenging as each digit and mathematical symbol is provided as a character and the expected output is also expected as characters. For example, the input `10+6` with the output `16` would be represented by the sequences:

```
Input: ['1', '0', '+', '6']
Output: ['1', '6']
```

Listing 9.5: Example of input and output sequences for the addition problem.

The model must learn not only the integer nature of the characters, but also the nature of the mathematical operation to perform. Notice how sequence is now important, and that randomly shuffling the input will create a nonsense sequence that could not be related to the output sequence. Also notice how the number of digits could vary in both the input and output sequences. Technically this makes the addition prediction problem a sequence-to-sequence problem that requires a many-to-many model to address.

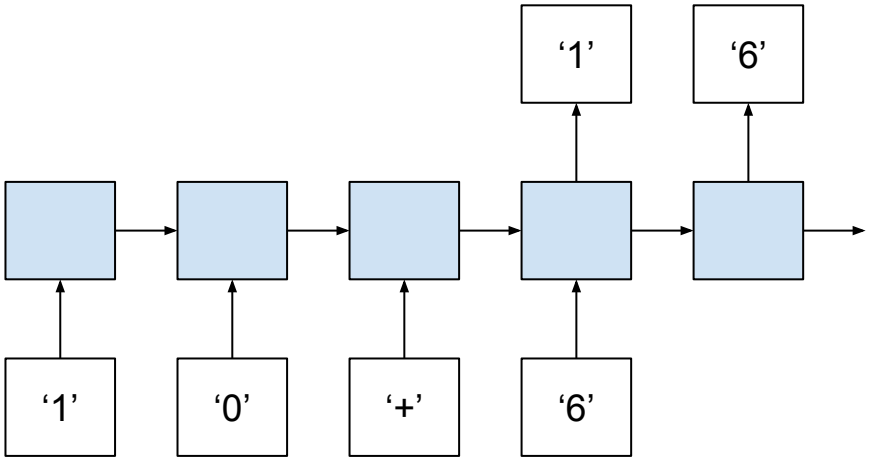

Figure 9.2: Addition prediction problem framed with a many-to-many prediction model.

We can keep things simple with addition of two numbers, but we can see how this may be scaled to a variable number of terms and mathematical operations that could be given as input for the model to learn and generalize. This problem can be implemented in Python. We will divide this into the following steps:

1. Generate Sum Pairs.
2. Integers to Padded Strings.
3. Integer Encoded Sequences.
4. One Hot Encoded Sequences.
5. Sequence Generation Pipeline.
6. Decode Sequences.

9.3.1 Generate Sum Pairs

The first step is to generate sequences of random integers and their sum. We can put this in a function named `random_sum_pairs()`, as follows.

```
from random import seed
from random import randint

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y
```

```
seed(1)
n_samples = 1
n_numbers = 2
largest = 10
# generate pairs
X, y = random_sum_pairs(n_samples, n_numbers, largest)
print(X, y)
```

Listing 9.6: Example of generating random a sequence pair.

Running just this function prints a single example of adding two random integers between 1 and 10.

```
[[3, 10]] [13]
```

Listing 9.7: Example of output of generating a random sequence pair.

9.3.2 Integers to Padded Strings

The next step is to convert the integers to strings. The input string will have the format `'10+10'` and the output string will have the format `'20'`. Key to this function is the padding of numbers to ensure that each input and output sequence has the same number of characters. A padding character should be different from the data so the model can learn to ignore them. In this case, we use the space character for padding(' ') and pad the string on the left, keeping the information on the far right.

There are other ways to pad, such as padding each term individually. Try it and see if it results in better performance. Padding requires we know how long the longest sequence may be. We can calculate this easily by taking the `log10()` of the largest integer we can generate and the ceiling of that number to get an idea of how many chars are needed for each number. We add 1 to the largest number to ensure we expect 3 chars instead of 2 chars for the case of a round largest number, like 200 and take the ceiling of the result (e.g. `ceil(log10(largest+1))`). We then need to add the right number of plus symbols (e.g. `n_numbers - 1`).

```
max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
```

Listing 9.8: Example of calculating the maximum length of input sequences.

We can make this concrete with a worked example where the total number of terms (`n_numbers`) is 3 and the largest value (`largest`) is 10.

```
max_length = n_numbers * ceil(log10(largest+1)) + n_numbers - 1
max_length = 3 * ceil(log10(10+1)) + 3 - 1
max_length = 3 * ceil(1.0413926851582251) + 3 - 1
max_length = 3 * 2 + 3 - 1
max_length = 6 + 3 - 1
max_length = 8
```

Listing 9.9: Worked example of maximum input sequence length.

Intuitively, we would expect 2 spaces for each term (e.g. `['1','0']`) multiplied by 3 terms, or a maximum length of input sequences of 6 spaces with two more spaces for the addition symbols (e.g. `['1','0','+','1','0','+','1','0']`) making the largest possible sequence 8 characters in length. This is what we see in the worked example.

A similar process is repeated on the output sequence, without the plus symbols, of course.

```
max_length = int(ceil(log10(n_numbers * (largest+1))))
```

Listing 9.10: Example of calculating the length of output sequences.

Again, we can make this concrete by calculating the expected maximum output sequence length for the above example with the total number of terms (`n_numbers`) is 3 and the largest value (`largest`) is 10.

```
max_length = ceil(log10(n_numbers * (largest+1)))
max_length = ceil(log10(3 * (10+1)))
max_length = ceil(log10(33))
max_length = ceil(1.5185139398778875)
max_length = 2
```

Listing 9.11: Worked example of maximum output sequence length.

Again, intuitively, we would expect the largest possible addition to be `10+10+10` or the value of `30`. This would require a maximum length of 2, and this is what we see in the worked example. The example below adds the `to_string()` function and demonstrates its usage with a single input/output pair.

```
from random import seed
from random import randint
from math import ceil
from math import log10

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y

# convert data to strings
def to_string(X, y, n_numbers, largest):
  max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
  Xstr = list()
  for pattern in X:
    strp = '+'.join([str(n) for n in pattern])
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    Xstr.append(strp)
  max_length = int(ceil(log10(n_numbers * (largest+1))))
  ystr = list()
  for pattern in y:
    strp = str(pattern)
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    ystr.append(strp)
  return Xstr, ystr

seed(1)
n_samples = 1
n_numbers = 2
largest = 10
```

```
# generate pairs
X, y = random_sum_pairs(n_samples, n_numbers, largest)
print(X, y)
# convert to strings
X, y = to_string(X, y, n_numbers, largest)
print(X, y)
```

Listing 9.12: Example of converting a sequence pair to padded characters.

Running this example first prints the integer sequence and the padded string representation of the same sequence.

```
[[3, 10]] [13]
[' 3+10'] ['13']
```

Listing 9.13: Example of output of converting a sequence pair to padded characters.

9.3.3 Integer Encoded Sequences

Next, we need to encode each character in the string as an integer value. We have to work with numbers in neural networks after all, not characters. Integer encoding transforms the problem into a classification problem where the output sequence may be considered class outputs with 11 possible values each. This just so happens to be integers with some ordinal relationship (the first 10 class values). To perform this encoding, we must define the full alphabet of symbols that may appear in the string encoding, as follows:

```
alphabet = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', ' ']
```

Listing 9.14: Example of defining a character alphabet.

Integer encoding then becomes a simple process of building a lookup table of character-to-integer offset and converting each char of each string, one by one. The example below provides the `integer_encode()` function for integer encoding and demonstrates how to use it.

```
from random import seed
from random import randint
from math import ceil
from math import log10

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y

# convert data to strings
def to_string(X, y, n_numbers, largest):
  max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
  Xstr = list()
  for pattern in X:
    strp = '+'.join([str(n) for n in pattern])
```

```
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    Xstr.append(strp)
  max_length = int(ceil(log10(n_numbers * (largest+1))))
  ystr = list()
  for pattern in y:
    strp = str(pattern)
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    ystr.append(strp)
  return Xstr, ystr

# integer encode strings
def integer_encode(X, y, alphabet):
  char_to_int = dict((c, i) for i, c in enumerate(alphabet))
  Xenc = list()
  for pattern in X:
    integer_encoded = [char_to_int[char] for char in pattern]
    Xenc.append(integer_encoded)
  yenc = list()
  for pattern in y:
    integer_encoded = [char_to_int[char] for char in pattern]
    yenc.append(integer_encoded)
  return Xenc, yenc

seed(1)
n_samples = 1
n_numbers = 2
largest = 10
# generate pairs
X, y = random_sum_pairs(n_samples, n_numbers, largest)
print(X, y)
# convert to strings
X, y = to_string(X, y, n_numbers, largest)
print(X, y)
# integer encode
alphabet = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', ' ']
X, y = integer_encode(X, y, alphabet)
print(X, y)
```

Listing 9.15: Example of integer encoding padded sequences.

Running the example prints the integer encoded version of each string encoded pattern. We can see that the space character (‘ ’) was encoded with 11 and the three character (‘3’) was encoded as 3, and so on.

```
[[3, 10]] [13]
[' 3+10'] ['13']
[[11, 3, 10, 1, 0]] [[1, 3]]
```

Listing 9.16: Example output from integer encoding input and output sequences.

9.3.4 One Hot Encoded Sequences

The next step is to binary encode the integer encoding sequences. This involves converting each integer to a binary vector with the same length as the alphabet and marking the specific integer with a 1. For example, a 0 integer represents the ‘0’ character and would be encoded as a

binary vector with a 1 in the 0th position of an 11 element vector: `[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`. The example below defines the `one_hot_encode()` function for binary encoding and demonstrates how to use it.

```
from random import seed
from random import randint
from math import ceil
from math import log10

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y

# convert data to strings
def to_string(X, y, n_numbers, largest):
  max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
  Xstr = list()
  for pattern in X:
    strp = '+'.join([str(n) for n in pattern])
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    Xstr.append(strp)
  max_length = int(ceil(log10(n_numbers * (largest+1))))
  ystr = list()
  for pattern in y:
    strp = str(pattern)
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    ystr.append(strp)
  return Xstr, ystr

# integer encode strings
def integer_encode(X, y, alphabet):
  char_to_int = dict((c, i) for i, c in enumerate(alphabet))
  Xenc = list()
  for pattern in X:
    integer_encoded = [char_to_int[char] for char in pattern]
    Xenc.append(integer_encoded)
  yenc = list()
  for pattern in y:
    integer_encoded = [char_to_int[char] for char in pattern]
    yenc.append(integer_encoded)
  return Xenc, yenc

# one hot encode
def one_hot_encode(X, y, max_int):
  Xenc = list()
  for seq in X:
    pattern = list()
    for index in seq:
      vector = [0 for _ in range(max_int)]
      vector[index] = 1
```

```
      pattern.append(vector)
    Xenc.append(pattern)
  yenc = list()
  for seq in y:
    pattern = list()
    for index in seq:
      vector = [0 for _ in range(max_int)]
      vector[index] = 1
      pattern.append(vector)
    yenc.append(pattern)
  return Xenc, yenc

seed(1)
n_samples = 1
n_numbers = 2
largest = 10
# generate pairs
X, y = random_sum_pairs(n_samples, n_numbers, largest)
print(X, y)
# convert to strings
X, y = to_string(X, y, n_numbers, largest)
print(X, y)
# integer encode
alphabet = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', ' ']
X, y = integer_encode(X, y, alphabet)
print(X, y)
# one hot encode
X, y = one_hot_encode(X, y, len(alphabet))
print(X, y)
```

Listing 9.17: Example of one hot encoding an integer encoded sequences.

Running the example prints the binary encoded sequence for each integer encoding. I've added some new lines to make the input and output binary encodings clearer. You can see that a single sum pattern becomes a sequence of 5 binary encoded vectors, each with 11 elements. The output, or sum, becomes a sequence of 2 binary encoded vectors, again each with 11 elements.

```
[[3, 10]] [13]
[' 3+10'] ['13']
[[11, 3, 10, 1, 0]] [[1, 3]]
[[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
  [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]]
 [[[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
   [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]]]
```

Listing 9.18: Example output from one hot encoding an integer encoded sequences.

9.3.5 Sequence Generation Pipeline

We can tie all of these steps together into a function called `generate_data()`, listed below. Given a designed number of samples, number of terms, the largest value of each term, and the alphabet of possible characters, the function will generate a set of input and output sequences.

```
# generate an encoded dataset
def generate_data(n_samples, n_numbers, largest, alphabet):
  # generate pairs
  X, y = random_sum_pairs(n_samples, n_numbers, largest)
  # convert to strings
  X, y = to_string(X, y, n_numbers, largest)
  # integer encode
  X, y = integer_encode(X, y, alphabet)
  # one hot encode
  X, y = one_hot_encode(X, y, len(alphabet))
  # return as NumPy arrays
  X, y = array(X), array(y)
  return X, y
```

Listing 9.19: Function for generating a sequence, encoding and reshaping it for an LSTM model.

9.3.6 Decode Sequences

Finally, we need to invert the encoding to convert the output vectors back into numbers so we can compare expected output integers to predicted integers. The `invert()` function below performs this operation. Key is first converting the binary encoding back into an integer using the `argmax()` function, then converting the integer back into a character using a reverse mapping of the integers to chars from the alphabet.

```
# invert encoding
def invert(seq, alphabet):
  int_to_char = dict((i, c) for i, c in enumerate(alphabet))
  strings = list()
  for pattern in seq:
    string = int_to_char[argmax(pattern)]
    strings.append(string)
  return ''.join(strings)
```

Listing 9.20: Function for deciding an encoded input or output sequence.

We now have everything we need to prepare data for this example.

9.4 Define and Compile the Model

The first step is to define the specifications of the sequence prediction problem. We must specify 3 parameters as input to the `generate_data()` function (above) for generating samples of input-output sequences:

- `n_terms`: The number of terms in the equation, (e.g. 2 for `10+10`).
- `largest`: The largest numerical value for each term (e.g. 10 for values between 1-10).
- `alphabet`: The symbols used to encode the input and output sequences (e.g. `0-9`, `+` and ‘ ’)

We will use a configuration of the problem that has a modest complexity. Each instance will be comprised of 3 terms with the maximum value of 10 per term. The alphabet remains fixed regardless of configuration with the values `0-9`, `'+'`, and `' '`.

```
# number of math terms
n_terms = 3
# largest value for any single input digit
largest = 10
# scope of possible symbols for each input or output time step
alphabet = [str(x) for x in range(10)] + ['+', ' ']
```

Listing 9.21: Example of configuring the problem instance.

The network needs three configuration values defined by the specification of the addition problem:

- `n_chars`: The size of the alphabet for a single time step (e.g. 12 for 0-9, '+' and ' ').
- `n_in_seq_length`: The number of time steps of encoded input sequences (e.g. 8 for '10+10+10').
- `n_out_seq_length`: The number of time steps of an encoded output sequence (e.g. 2 for '30')

The `n_chars` variable is used to define the number of features in the input layer and the number of features in the output layer for each input and output time step. The `n_in_seq_length` variable is used to define the number of time steps for the input layer of the network. The `n_out_seq_length` variable is used to define the number of times to repeat the encoded input in the `RepeatVector` that in turn defines the length of the sequence fed to the decoder for creating the output sequence. The definition of `n_in_seq_length` and `n_out_seq_length` uses the same code from the `to_string()` function used to map the integer sequence to strings.

```
# size of alphabet: (12 for 0-9, + and ' ')
n_chars = len(alphabet)
# length of encoded input sequence (8 for '10+10+10)
n_in_seq_length = int(n_terms * ceil(log10(largest+1)) + n_terms - 1)
# length of encoded output sequence (2 for '30')
n_out_seq_length = int(ceil(log10(n_terms * (largest+1))))
```

Listing 9.22: Example of defining network configuration based on the problem instance.

We are now ready to define the Encoder-Decoder LSTM. We will use a single `LSTM` layer for the encoder and another single layer for the decoder. The encoder is defined with 75 memory cells and the decoder with 50 memory cells. The number of memory cells was found with a little trial and error. The asymmetry in layer sizes in the encoder and decoder seems like a natural organization given that input sequences are relatively longer than output sequences.

The output layer uses the categorical log loss for the 12 possible *classes* that may be predicted. The efficient Adam implementation of gradient descent is used and accuracy will be calculated during training and model evaluation.

```
# define LSTM
model = Sequential()
model.add(LSTM(75, input_shape=(n_in_seq_length, n_chars)))
model.add(RepeatVector(n_out_seq_length))
```

```
model.add(LSTM(50, return_sequences=True))
model.add(TimeDistributed(Dense(n_chars, activation='softmax')))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Listing 9.23: Example of defining and compiling the Encoder-Decoder LSTM.

Running the example prints a summary of the network structure. We can see that the Encoder will output a fixed size vector with the length of 75 for a given input sequence. This sequence is repeated 2 times to provide a sequence of 2 time steps of 75 *features* to the decoder. The decoder outputs two time steps of 50 features to the `Dense` output layer that processes these one at a time via the `TimeDistributed` wrapper to output one encoded character at a time.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 75)                26400
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 2, 75)             0
_________________________________________________________________
lstm_2 (LSTM)                (None, 2, 50)             25200
_________________________________________________________________
time_distributed_1 (TimeDist (None, 2, 12)             612
=================================================================
Total params: 52,212
Trainable params: 52,212
Non-trainable params: 0
_________________________________________________________________
```

Listing 9.24: Example output from defining and compiling the Encoder-Decoder LSTM.

9.5 Fit the Model

The model is fit on a single epoch of 75,000 randomly generated instances of input-output pairs. The number of sequences is a proxy for the number of training epochs. The total of 75,000 and a batch size of 32 were found with a little trial and error and are by no means an optimal configuration.

```
# fit LSTM
X, y = generate_data(75000, n_terms, largest, alphabet)
model.fit(X, y, epochs=1, batch_size=32)
```

Listing 9.25: Example of fitting the defined Encoder-Decoder LSTM.

Fitting the model provides a progress bar that shows the loss and accuracy of the model at the end of each batch. The model does not take long to fit on the CPU. If the progress bar interferes with your development environment, you can turn it off by setting `verbose=0` in the call to the `fit()` function.

```
75000/75000 [==============================] - 37s - loss: 0.6982 - acc: 0.7943
```

Listing 9.26: Example output from fitting the defined Encoder-Decoder LSTM.

9.6 Evaluate the Model

We can evaluate the model by generating predictions on 100 different randomly generated input-output pairs. The result will give an estimate of the model skill on randomly generated examples in general.

```
# evaluate LSTM
X, y = generate_data(100, n_terms, largest, alphabet)
loss, acc = model.evaluate(X, y, verbose=0)
print('Loss: %f, Accuracy: %f' % (loss, acc*100))
```

Listing 9.27: Example of evaluating the fit Encoder-Decoder LSTM.

Running the example prints both the log loss and accuracy of the model. Your specific values may differ because of the stochastic nature of neural networks, but the model accuracy should be in the high 90s.

```
Loss: 0.128379, Accuracy: 100.000000
```

Listing 9.28: Example output from evaluating the fit Encoder-Decoder LSTM.

9.7 Make Predictions with the Model

We can make predictions using the fit model. We will demonstrate making one prediction at a time and provide a summary of the decoded input, expected output, and predicted output. Printing the decoded output gives us a more concrete connection to the problem and model performance. Here, we generate 10 new random input-output sequence pairs, make a prediction using the fit model for each, decode all the sequences involved, and print them to the screen.

```
# predict
for _ in range(10):
  # generate an input-output pair
  X, y = generate_data(1, n_terms, largest, alphabet)
  # make prediction
  yhat = model.predict(X, verbose=0)
  # decode input, expected and predicted
  in_seq = invert(X[0], alphabet)
  out_seq = invert(y[0], alphabet)
  predicted = invert(yhat[0], alphabet)
  print('%s = %s (expect %s)' % (in_seq, predicted, out_seq))
```

Listing 9.29: Example of making predictions with the fit Encoder-Decoder LSTM.

Running the example shows that the model gets most of the sequences correct. The specific sequences you generate and the skill of the model on just ten examples will vary. Try running the prediction piece a few times to get a good feel for the model behavior.

```
  9+10+9 = 27 (expect 28)
   9+6+9 = 24 (expect 24)
  8+9+10 = 27 (expect 27)
  9+9+10 = 28 (expect 28)
   2+4+5 = 11 (expect 11)
   2+9+7 = 18 (expect 18)
   7+3+2 = 12 (expect 12)
```

```
  4+1+4 = 9 (expect 9)
  8+6+7 = 21 (expect 21)
  5+2+7 = 14 (expect 14)
```

Listing 9.30: Example output from making predictions with the fit Encoder-Decoder LSTM.

9.8 Complete Example

For completeness, the full code listing is provided below for your reference.

```
from random import randint
from numpy import array
from math import ceil
from math import log10
from numpy import argmax
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import TimeDistributed
from keras.layers import RepeatVector

# generate lists of random integers and their sum
def random_sum_pairs(n_examples, n_numbers, largest):
  X, y = list(), list()
  for _ in range(n_examples):
    in_pattern = [randint(1,largest) for _ in range(n_numbers)]
    out_pattern = sum(in_pattern)
    X.append(in_pattern)
    y.append(out_pattern)
  return X, y

# convert data to strings
def to_string(X, y, n_numbers, largest):
  max_length = int(n_numbers * ceil(log10(largest+1)) + n_numbers - 1)
  Xstr = list()
  for pattern in X:
    strp = '+'.join([str(n) for n in pattern])
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    Xstr.append(strp)
  max_length = int(ceil(log10(n_numbers * (largest+1))))
  ystr = list()
  for pattern in y:
    strp = str(pattern)
    strp = ''.join([' ' for _ in range(max_length-len(strp))]) + strp
    ystr.append(strp)
  return Xstr, ystr

# integer encode strings
def integer_encode(X, y, alphabet):
  char_to_int = dict((c, i) for i, c in enumerate(alphabet))
  Xenc = list()
  for pattern in X:
    integer_encoded = [char_to_int[char] for char in pattern]
    Xenc.append(integer_encoded)
```

```
  yenc = list()
  for pattern in y:
    integer_encoded = [char_to_int[char] for char in pattern]
    yenc.append(integer_encoded)
  return Xenc, yenc

# one hot encode
def one_hot_encode(X, y, max_int):
  Xenc = list()
  for seq in X:
    pattern = list()
    for index in seq:
      vector = [0 for _ in range(max_int)]
      vector[index] = 1
      pattern.append(vector)
    Xenc.append(pattern)
  yenc = list()
  for seq in y:
    pattern = list()
    for index in seq:
      vector = [0 for _ in range(max_int)]
      vector[index] = 1
      pattern.append(vector)
    yenc.append(pattern)
  return Xenc, yenc

# generate an encoded dataset
def generate_data(n_samples, n_numbers, largest, alphabet):
  # generate pairs
  X, y = random_sum_pairs(n_samples, n_numbers, largest)
  # convert to strings
  X, y = to_string(X, y, n_numbers, largest)
  # integer encode
  X, y = integer_encode(X, y, alphabet)
  # one hot encode
  X, y = one_hot_encode(X, y, len(alphabet))
  # return as numpy arrays
  X, y = array(X), array(y)
  return X, y

# invert encoding
def invert(seq, alphabet):
  int_to_char = dict((i, c) for i, c in enumerate(alphabet))
  strings = list()
  for pattern in seq:
    string = int_to_char[argmax(pattern)]
    strings.append(string)
  return ''.join(strings)

# configure problem

# number of math terms
n_terms = 3
# largest value for any single input digit
largest = 10
# scope of possible symbols for each input or output time step
```

```
alphabet = [str(x) for x in range(10)] + ['+', ' ']

# size of alphabet: (12 for 0-9, + and ' ')
n_chars = len(alphabet)
# length of encoded input sequence (8 for '10+10+10)
n_in_seq_length = int(n_terms * ceil(log10(largest+1)) + n_terms - 1)
# length of encoded output sequence (2 for '30')
n_out_seq_length = int(ceil(log10(n_terms * (largest+1))))

# define LSTM
model = Sequential()
model.add(LSTM(75, input_shape=(n_in_seq_length, n_chars)))
model.add(RepeatVector(n_out_seq_length))
model.add(LSTM(50, return_sequences=True))
model.add(TimeDistributed(Dense(n_chars, activation='softmax')))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()

# fit LSTM
X, y = generate_data(75000, n_terms, largest, alphabet)
model.fit(X, y, epochs=1, batch_size=32)

# evaluate LSTM
X, y = generate_data(100, n_terms, largest, alphabet)
loss, acc = model.evaluate(X, y, verbose=0)
print('Loss: %f, Accuracy: %f' % (loss, acc*100))

# predict
for _ in range(10):
  # generate an input-output pair
  X, y = generate_data(1, n_terms, largest, alphabet)
  # make prediction
  yhat = model.predict(X, verbose=0)
  # decode input, expected and predicted
  in_seq = invert(X[0], alphabet)
  out_seq = invert(y[0], alphabet)
  predicted = invert(yhat[0], alphabet)
  print('%s = %s (expect %s)' % (in_seq, predicted, out_seq))
```

Listing 9.31: Complete examples of the Encoder-Decoder LSTM model on the Addition Prediction prediction problem.

9.9 Further Reading

This section provides some resources for further reading.

9.9.1 Papers on Encoder-Decoder LSTM

- *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.
  https://arxiv.org/abs/1406.1078

- *Sequence to Sequence Learning with Neural Networks*, 2014.
  https://arxiv.org/abs/1409.3215
- *Show and Tell: A Neural Image Caption Generator*, 2014.
  https://arxiv.org/abs/1411.4555
- *Learning to Execute*, 2015.
  http://arxiv.org/abs/1410.4615
- *A Neural Conversational Model*, 2015.
  https://arxiv.org/abs/1506.05869

9.9.2 Keras API

- `RepeatVector` Keras API.
  https://keras.io/layers/core/#repeatvector
- `TimeDistributed` Keras API.
  https://keras.io/layers/wrappers/#timedistributed

9.10 Extensions

Do you want to dive deeper into Encoder-Decoder LSTMs? This section lists some challenging extensions to this lesson.

- List 10 sequence-to-sequence prediction problems that may benefit from the Encoder-Decoder LSTM architecture.
- Increase the number of terms or number of digits and tune the model to get 100% accuracy.
- Design a study to compare model size to problem complexity (terms and/or digits).
- Update the example to support a variable number of terms in a given instance and tune a model to get 100% accuracy.
- Add support for other math operations like subtraction, division, and multiplication.

Post your extensions online and share the link with me; I'd love to see what you come up with!

9.11 Summary

In this lesson, you discovered how to develop an Encoder-Decoder LSTM model. Specifically, you learned:

- The Encoder-Decoder LSTM architecture and how to implement it in Keras.
- The addition sequence-to-sequence prediction problem.

- How to develop an Encoder-Decoder LSTM for the addition seq2seq prediction problem.

In the next lesson, you will discover how to develop and evaluate the Bidirectional LSTM model.

Chapter 10

How to Develop Bidirectional LSTMs

10.0.1 Lesson Goal

The goal of this lesson is to learn how to develop Bidirectional LSTM models. After completing this lesson, you will know:

- The Bidirectional LSTM architecture and how to implement it in Keras.
- The cumulative sum prediction problem.
- How to develop a Bidirectional LSTM for the cumulative sum prediction problem.

10.0.2 Lesson Overview

This lesson is divided into 7 parts; they are:

1. The Bidirectional LSTM.
2. Cumulative Sum Prediction Problem.
3. Define and Compile the Model.
4. Fit the Model.
5. Evaluate the Model.
6. Make Predictions With the Model.
7. Complete Example.

Let's get started.

10.1 The Bidirectional LSTM

10.1.1 Architecture

We have seen the benefit of reversing the order of input sequences for LSTMs discussed in the introduction of Encoder-Decoder LSTMs.

> We were surprised by the extent of the improvement obtained by reversing the words in the source sentences.
>
> — *Sequence to Sequence Learning with Neural Networks*, 2014.

Bidirectional LSTMs focus on the problem of getting the most out of the input sequence by stepping through input time steps in both the forward and backward directions. In practice, this architecture involves duplicating the first recurrent layer in the network so that there are now two layers side-by-side, then providing the input sequence as-is as input to the first layer and providing a reversed copy of the input sequence to the second. This approach was developed some time ago as a general approach for improving the performance of Recurrent Neural Networks (RNNs).

> To overcome the limitations of a regular RNN ... we propose a bidirectional recurrent neural network (BRNN) that can be trained using all available input information in the past and future of a specific time frame. ... The idea is to split the state neurons of a regular RNN in a part that is responsible for the positive time direction (forward states) and a part for the negative time direction (backward states)
>
> — *Bidirectional Recurrent Neural Networks*, 1997.

This approach has been used to great effect with LSTM Recurrent Neural Networks. Providing the entire sequence both forwards and backwards is based on the assumption that the whole sequence is available. This is generally a requirement in practice when using vectorized inputs. Nevertheless, it may raise a philosophical concern where ideally time steps are provided in order and just-in-time. The use of providing an input sequence bi-directionally was justified in the domain of speech recognition because there is evidence that in humans, the context of the whole utterance is used to interpret what is being said rather than a linear interpretation.

> ... relying on knowledge of the future seems at first sight to violate causality. How can we base our understanding of what we've heard on something that hasn't been said yet? However, human listeners do exactly that. Sounds, words, and even whole sentences that at first mean nothing are found to make sense in the light of future context. What we must remember is the distinction between tasks that are truly online - requiring an output after every input - and those where outputs are only needed at the end of some input segment.
>
> — *Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures*, 2005.

Although Bidirectional LSTMs were developed for speech recognition, the use of Bidirectional input sequences is now a staple of sequence prediction with LSTMs as an approach for lifting model performance.

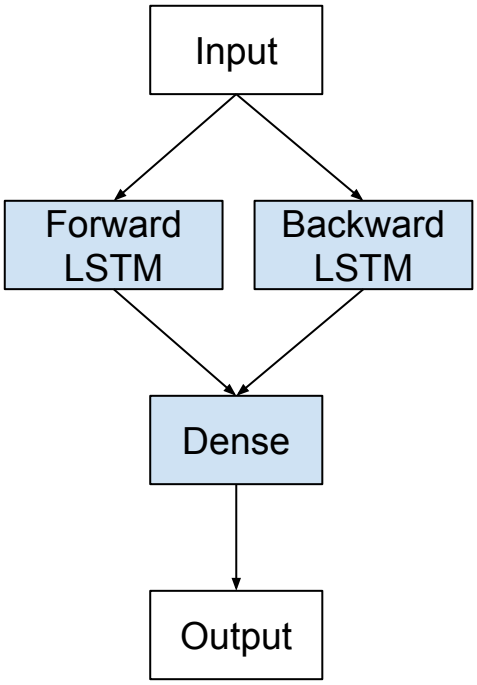

Figure 10.1: Bidirectional LSTM Architecture.

10.1.2 Implementation

The `LSTM` layer in Keras allow you to specify the directionality of the input sequence. This can be done by setting the `go_backwards` argument to `True` (defaults to `False`).

```
model = Sequential()
model.add(LSTM(..., input_shape=(...), go_backwards=True))
...
```

Listing 10.1: Example of a Vanilla LSTM model with backward input sequences.

Bidirectional LSTMs are a small step on top of this capability. Specifically, Bidirectional LSTMs are supported in Keras via the `Bidirectional` layer wrapper that essentially merges the output from two parallel LSTMs, one with input processed forward and one with output processed backwards. This wrapper takes a recurrent layer (e.g. the first hidden `LSTM` layer) as an argument.

```
model = Sequential()
model.add(Bidirectional(LSTM(...), input_shape=(...)))
...
```

Listing 10.2: Example of a `Bidirectional` wrapped `LSTM` layer.

The `Bidirectional` wrapper layer also allows you to specify the merge mode; that is how the forward and backward outputs should be combined before being passed on to the next layer. The options are:

- '`sum`': The outputs are added together.
- '`mul`': The outputs are multiplied together.

- 'concat': The outputs are concatenated together (the default), providing double the number of outputs to the next layer.
- 'ave': The average of the outputs is taken.

The default mode is to concatenate, and this is the method often used in studies of bidirectional LSTMs. In general, it might be a good idea to test each of the merge modes on your problem to see if you can improve upon the concatenate default option.

10.2 Cumulative Sum Prediction Problem

We will define a simple sequence classification problem to explore bidirectional LSTMs called the cumulative sum prediction problem. This section is divided into the following parts:

1. Cumulative Sum.
2. Sequence Generation.
3. Generate Multiple Sequences.

10.2.1 Cumulative Sum

The problem is defined as a sequence of random values between 0 and 1. This sequence is taken as input for the problem with each number provided once per time step. A binary label (0 or 1) is associated with each input. The output values are all 0. Once the cumulative sum of the input values in the sequence exceeds a threshold, then the output value flips from 0 to 1.

A threshold of one quarter ($\frac{1}{4}$) the sequence length is used. For example, below is a sequence of 10 input time steps (X):

```
0.63144003 0.29414551 0.91587952 0.95189228 0.32195638 0.60742236 0.83895793 0.18023048
    0.84762691 0.29165514
```

Listing 10.3: Example input sequence of random real values.

The corresponding classification output (y) would be:

```
0 0 0 1 1 1 1 1 1 1
```

Listing 10.4: Example output sequence of cumulative sum values.

We will frame the problem to make the best use of the Bidirectional LSTM architecture. The output sequence will be produced after the entire input sequence has been fed into the model. Technically, this means this is a sequence-to-sequence prediction problem that requires a many-to-many prediction model. It is also the case that the input and output sequences have the same number of time steps (length).

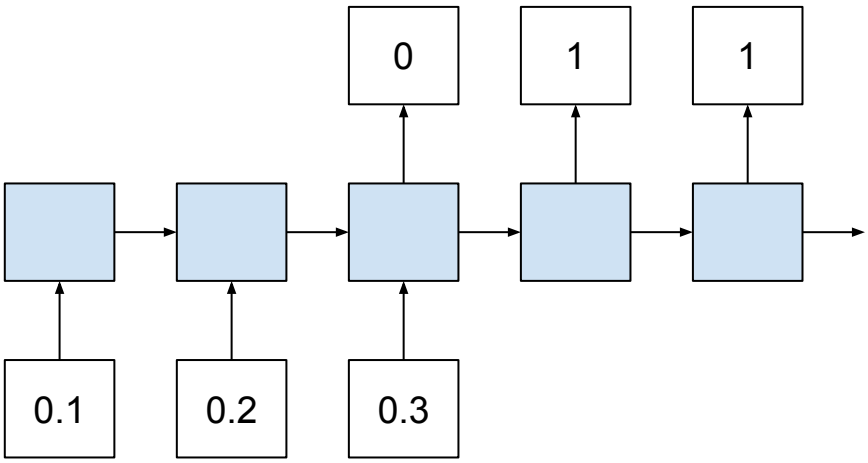

Figure 10.2: Cumulative sum prediction problem framed with a many-to-many prediction model.

10.2.2 Sequence Generation

We can implement this in Python. The first step is to generate a sequence of random values. We can use the `random()` function from the random module.

```
# create a sequence of random numbers in [0,1]
X = array([random() for _ in range(10)])
```

Listing 10.5: Example creating an input sequence of random real values.

We can define the threshold as one-quarter the length of the input sequence.

```
# calculate cut-off value to change class values
limit = 10/4.0
```

Listing 10.6: Example of calculating the cumulative sum threshold.

The cumulative sum of the input sequence can be calculated using the `cumsum()` NumPy function. This function returns a sequence of cumulative sum values, e.g.:

```
pos1, pos1+pos2, pos1+pos2+pos3, ...
```

Listing 10.7: Example of calculating a cumulative sum output sequence.

We can then calculate the output sequence as to whether each cumulative sum value exceeded the threshold.

```
# determine the class outcome for each item in cumulative sequence
y = array([0 if x < limit else 1 for x in cumsum(X)])
```

Listing 10.8: Example of implementing the calculating of the cumulative sum threshold.

The function below, named `get_sequence()`, draws all of this together, taking as input the length of the sequence, and returns the `X` and `y` components of a new problem case.

```
# create a sequence classification instance
def get_sequence(n_timesteps):
  # create a sequence of random numbers in [0,1]
  X = array([random() for _ in range(n_timesteps)])
```

```
  # calculate cut-off value to change class values
  limit = n_timesteps/4.0
  # determine the class outcome for each item in cumulative sequence
  y = array([0 if x < limit else 1 for x in cumsum(X)])
  return X, y
```

Listing 10.9: Function to create a random input and output sequence.

We can test this function with a new 10-step sequence as follows:

```
from random import random
from numpy import array
from numpy import cumsum

# create a cumulative sum sequence
def get_sequence(n_timesteps):
  # create a sequence of random numbers in [0,1]
  X = array([random() for _ in range(n_timesteps)])
  # calculate cut-off value to change class values
  limit = n_timesteps/4.0
  # determine the class outcome for each item in cumulative sequence
  y = array([0 if x < limit else 1 for x in cumsum(X)])
  return X, y

X, y = get_sequence(10)
print(X)
print(y)
```

Listing 10.10: Example of generating a random input and output sequence.

Running the example first prints the generated input sequence followed by the matching output sequence.

```
[ 0.22228819 0.26882207 0.069623 0.91477783 0.02095862 0.71322527
0.90159654 0.65000306 0.88845226 0.4037031 ]
[0 0 0 0 0 0 1 1 1 1]
```

Listing 10.11: Example output from generating a random input and output sequence.

10.2.3 Generate Multiple Sequences

We can define a function to create multiple sequences. The function below named `get_sequences()` takes the number of sequences to generate and the number of time steps per sequence as arguments and calls `get_sequence()` to generate the sequences. Once the specified number of sequences have been generated, the list of input and output sequences are reshaped to be three-dimensional and suitable for use with LSTMs.

```
# create multiple samples of cumulative sum sequences
def get_sequences(n_sequences, n_timesteps):
  seqX, seqY = list(), list()
  # create and store sequences
  for _ in range(n_sequences):
    X, y = get_sequence(n_timesteps)
    seqX.append(X)
    seqY.append(y)
  # reshape input and output for lstm
```

```
  seqX = array(seqX).reshape(n_sequences, n_timesteps, 1)
  seqY = array(seqY).reshape(n_sequences, n_timesteps, 1)
  return seqX, seqY
```

Listing 10.12: Function to generate sequences and format them for LSTM models.

We are now ready to start developing a Bidirectional LSTM for this problem.

10.3 Define and Compile the Model

First, we can define the complexity of the problem. We will limit the number of input time steps to a modest size; in this case, 10. This means the input shape will be 10 time steps with 1 feature.

```
# define problem
n_timesteps = 10
```

Listing 10.13: Example of configuring the problem.

Next, we need to define the hidden `LSTM` layer wrapped in a `Bidirectional` layer. We will use 50 memory cells in the `LSTM` hidden layer. The `Bidirectional` wrapper will double this, creating a second layer parallel to the first, also with 50 memory cells.

```
model.add(Bidirectional(LSTM(50, return_sequences=True), input_shape=(n_timesteps, 1)))
```

Listing 10.14: Example of adding the `Bidirectional` input layer.

A vector of 50 output values from each of the forward and backward `LSTM` hidden layers will be concatenated (the default merge method of the `Bidirectional` wrapper layer) to create a 100 element vector output. This is provided as input to a `Dense` layer that is wrapped in a `TimeDistributed` layer. This has the effect of reusing the weights of the `Dense` layer in order to create each output time step.

```
model.add(TimeDistributed(Dense(1, activation='sigmoid')))
```

Listing 10.15: Example of adding the `TimeDistributed` output layer.

The `Bidirectional` LSTM layers return sequences to the `TimeDistributed` wrapped `Dense` layer. This has the effect of providing one concatenated 100 element vector to the `Dense` layer as input for each output time step. If a `TimeDistributed` wrapper was not used, a single 100 element vector would be provided to the `Dense` layer from which it would be required to output 10 time steps of classification. This would seem to be a more challenging problem for the model.

Putting this together, the model is defined below. The `sigmoid` activation is used in the `Dense` output layer and the binary log loss is optimized as each output time step is a binary classification of whether or not the cumulative sum threshold has been exceeded. The Adam implementation of gradient descent is used to optimize the weights and the classification accuracy is calculated during model training and evaluation.

```
# define LSTM
model = Sequential()
model.add(Bidirectional(LSTM(50, return_sequences=True), input_shape=(n_timesteps, 1)))
model.add(TimeDistributed(Dense(1, activation='sigmoid')))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

Listing 10.16: Example of defining and compiling the Bidirectional LSTM model.

Running this code prints a summary of the compiled model. We can confirm that the `Dense` layer has 100 weights (plus the bias), one for each item in the 100 element concatenated vector provided from the `Bidirectional` wrapped `LSTM` hidden layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional_1 (Bidirection (None, 10, 100)      20800
_________________________________________________________________
time_distributed_1 (TimeDist (None, 10, 1)        101
=================================================================
Total params: 20,901
Trainable params: 20,901
Non-trainable params: 0
_________________________________________________________________
```

Listing 10.17: Example output from defining and compiling the Bidirectional LSTM model.

10.4 Fit the Model

We can use the `get_sequences()` function to generate a large number of random examples on which to fit the model. We can simplify training by using the number of randomly generated sequences as a proxy for epochs. This allows us to generate a large number of examples, in this case 50,000, store them in memory, and fit them in one Keras epoch.

The batch size of 10 is used to balance learning speed and computational efficiency. Both the number of samples and batch size were found with some trial and error. Experiment with different values and see if you can train an accurate model with less computational effort.

```
# train LSTM
X, y = get_sequences(50000, n_timesteps)
model.fit(X, y, epochs=1, batch_size=10)
```

Listing 10.18: Example of fitting a compiled the Bidirectional LSTM model.

Fitting the model does not take long. A progress bar is provided during training and the log loss and model accuracy are updated each batch.

```
50000/50000 [==============================] - 97s - loss: 0.0508 - acc: 0.9817
```

Listing 10.19: Example output from fitting a compiled the Bidirectional LSTM model.

10.5 Evaluate the Model

We can evaluate the model by generating 100 new random sequences and calculating the accuracy of the predictions made by the fit model.

```
# evaluate LSTM
X, y = get_sequences(100, n_timesteps)
```

```
loss, acc = model.evaluate(X, y, verbose=0)
print('Loss: %f, Accuracy: %f' % (loss, acc*100))
```

Listing 10.20: Example of evaluating a fit the Bidirectional LSTM model.

Running this example prints both the log loss and accuracy. We can see that the model achieves 100% accuracy. The accuracy may vary when you run the example given the stochastic nature of the algorithms. You should see model skill in the high 90s. Try running the example a few times.

```
Loss: 0.016752, Accuracy: 100.000000
```

Listing 10.21: Example output from evaluating a fit the Bidirectional LSTM model.

10.6 Make Predictions with the Model

We can make predictions in a similar way as evaluating the model. In this case, we will generate 10 new random sequences, make predictions for each, and compare the predicted output sequence to the expected output sequence.

```
# make predictions
for _ in range(10):
  X, y = get_sequences(1, n_timesteps)
  yhat = model.predict_classes(X, verbose=0)
  exp, pred = y.reshape(n_timesteps), yhat.reshape(n_timesteps)
  print('y=%s, yhat=%s, correct=%s' % (exp, pred, array_equal(exp,pred)))
```

Listing 10.22: Example of making predictions with a fit the Bidirectional LSTM model.

Running the example prints both the expected (`y`) and predicted (`yhat`) output sequences and whether or not the predicted sequence was correct. We can see that, at least in this case, 2 of the 10 sequences were predicted with an error at one time step.

Your specific results will vary, but you should see similar behavior on average. This is a challenging problem, and even for a model that fits a large number of examples and shows good accuracy, it can still make errors in predicting new sequences.

```
y=[0 0 0 0 0 0 1 1 1 1], yhat=[0 0 0 0 0 0 1 1 1 1], correct=True
y=[0 0 0 0 1 1 1 1 1 1], yhat=[0 0 0 0 1 1 1 1 1 1], correct=True
y=[0 0 0 1 1 1 1 1 1 1], yhat=[0 0 0 1 1 1 1 1 1 1], correct=True
y=[0 0 0 0 0 0 0 1 1 1], yhat=[0 0 0 0 0 0 0 0 1 1], correct=False
y=[0 0 0 0 0 1 1 1 1 1], yhat=[0 0 0 0 0 1 1 1 1 1], correct=True
y=[0 0 0 1 1 1 1 1 1 1], yhat=[0 0 0 1 1 1 1 1 1 1], correct=True
y=[0 0 0 0 0 1 1 1 1 1], yhat=[0 0 0 0 0 0 1 1 1 1], correct=False
y=[0 0 0 0 1 1 1 1 1 1], yhat=[0 0 0 0 1 1 1 1 1 1], correct=True
y=[0 0 0 0 0 0 0 0 1 1], yhat=[0 0 0 0 0 0 0 0 1 1], correct=True
y=[0 0 0 1 1 1 1 1 1 1], yhat=[0 0 0 1 1 1 1 1 1 1], correct=True
```

Listing 10.23: Example output from making predictions with a fit the Bidirectional LSTM model.

10.7 Complete Example

The full example is listed below for completeness and your reference.

```
from random import random
from numpy import array
from numpy import cumsum
from numpy import array_equal
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import TimeDistributed
from keras.layers import Bidirectional

# create a cumulative sum sequence
def get_sequence(n_timesteps):
  # create a sequence of random numbers in [0,1]
  X = array([random() for _ in range(n_timesteps)])
  # calculate cut-off value to change class values
  limit = n_timesteps/4.0
  # determine the class outcome for each item in cumulative sequence
  y = array([0 if x < limit else 1 for x in cumsum(X)])
  return X, y

# create multiple samples of cumulative sum sequences
def get_sequences(n_sequences, n_timesteps):
  seqX, seqY = list(), list()
  # create and store sequences
  for _ in range(n_sequences):
    X, y = get_sequence(n_timesteps)
    seqX.append(X)
    seqY.append(y)
  # reshape input and output for lstm
  seqX = array(seqX).reshape(n_sequences, n_timesteps, 1)
  seqY = array(seqY).reshape(n_sequences, n_timesteps, 1)
  return seqX, seqY

# define problem
n_timesteps = 10

# define LSTM
model = Sequential()
model.add(Bidirectional(LSTM(50, return_sequences=True), input_shape=(n_timesteps, 1)))
model.add(TimeDistributed(Dense(1, activation='sigmoid')))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()

# train LSTM
X, y = get_sequences(50000, n_timesteps)
model.fit(X, y, epochs=1, batch_size=10)

# evaluate LSTM
X, y = get_sequences(100, n_timesteps)
loss, acc = model.evaluate(X, y, verbose=0)
print('Loss: %f, Accuracy: %f' % (loss, acc*100))

# make predictions
for _ in range(10):
  X, y = get_sequences(1, n_timesteps)
```

```
yhat = model.predict_classes(X, verbose=0)
exp, pred = y.reshape(n_timesteps), yhat.reshape(n_timesteps)
print('y=%s, yhat=%s, correct=%s' % (exp, pred, array_equal(exp,pred)))
```

Listing 10.24: Complete example of the Bidirectional LSTM on the Cumulative Sum problem.

10.8 Further Reading

This section provides some resources for further reading.

10.8.1 Research Papers

- *Bidirectional Recurrent Neural Networks*, 1997.
- *Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures*, 2005.
- *Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition*, 2005.
- *Speech Recognition with Deep Recurrent Neural Networks*, 2013.
  https://arxiv.org/abs/1303.5778

10.8.2 APIs

- `random()` Python API.
  https://docs.python.org/3/library/random.html
- `cumsum()` NumPy API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.cumsum.html
- `Bidirectional` Keras API.
  https://keras.io/layers/wrappers/#bidirectional

10.9 Extensions

Do you want to dive deeper into Bidirectional LSTMs? This section lists some challenging extensions to this lesson.

- List 5 examples of sequence prediction problems that may benefit from a Bidirectional LSTM.
- Tune the number of memory cells, training examples, and batch size to develop a smaller or faster trained model with 100% accuracy.
- Design and execute an experiment to compare model size to problem complexity (e.g. sequence length).
- Design and execute an experiment to compare forward, backward, and bidirectional LSTM input direction.

- Design and execute an experiment to compare the combination methods for the Bidirectional LSTM wrapper layer.

Post your extensions online and share the link with me; I'd love to see what you come up with!

10.10 Summary

In this lesson, you discovered how to develop a Bidirectional LSTM model. Specifically, you learned:

- The Bidirectional LSTM architecture and how to implement it in Keras.
- The cumulative sum prediction problem.
- How to develop a Bidirectional LSTM for the cumulative sum prediction problem.

In the next lesson, you will discover how to develop and evaluate the Generative LSTM model.

Chapter 11

How to Develop Generative LSTMs

11.0.1 Lesson Goal

The goal of this lesson is to learn how to develop LSTMs for use as generative models. After completing this lesson, you will know:

- How LSTMs can be used as sequence generation models.
- How to frame shape drawing as a sequence generation problem.
- How to develop an LSTM to generate shapes.

11.0.2 Lesson Overview

This lesson is divided into 7 parts; they are:

1. The Generative LSTM.
2. Shape Generation Problem.
3. Define and Compile the Model.
4. Fit the Model.
5. Make Predictions With the Model.
6. Evaluate the Model.
7. Complete Example.

Let's get started.

11.1 The Generative LSTM

11.1.1 Generative Model

LSTMs can be used as a generative model. Given a large corpus of sequence data, such as text documents, LSTM models can be designed to learn the general structural properties of the corpus, and when given a seed input, can generate new sequences that are representative of the original corpus.

The problem of developing a model to generalize a corpus of text is called language modeling in the field of natural language processing. A language model may work at the word level and learn the probabilistic relationships between words in a document in order to accurately complete a sentence and generate entirely new sentences. At its most challenging, language models work at the character level, learning from sequences of characters, and generating new sequences one character at a time.

> The goal of character-level language modeling is to predict the next character in a sequence.
>
> — *Generating Text with Recurrent Neural Networks*, 2011.

Although more challenging, the added flexibility of a character-level model allows new words to be generated, punctuation added, and the generation of any other structures that may exist in the text data.

> ... predicting one character at a time is more interesting from the perspective of sequence generation, because it allows the network to invent novel words and strings.
>
> — *Generating Sequences With Recurrent Neural Networks*, 2013.

Language modeling is by far the most studied application of Generative LSTMs, perhaps because of the use of standard datasets where model performance can be quantified and compared. This approach has been used to generate text on a suite of interesting language modeling problems, such as:

- Generating Wikipedia articles (including markup).
- Generating snippets from great authors like Shakespeare.
- Generating technical manuscripts (including markup).
- Generating computer source code.
- Generating article headlines.

The quality of the results vary; for example, the markup or source code may require manual intervention to render or compile. Nevertheless, the results are impressive. The approach has also been applied to different domains where a large corpus of existing sequence information is available and new sequences can be generated one step at a time, such as:

- Handwriting generation.
- Music generation.
- Speech generation.

11.1.2 Architecture and Implementation

A Generative LSTM is not really architecture, it is more a change in perspective about what an LSTM predictive model learns and how the model is used. We could conceivably use any LSTM architecture as a generative model. In this case, we will use a simple Vanilla LSTM.

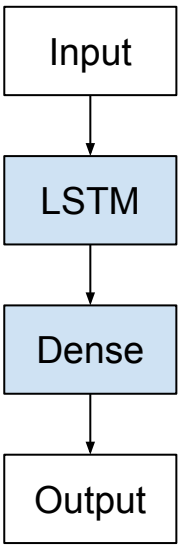

Figure 11.1: Generative LSTM Architecture, in this case the Vanilla LSTM.

In the case of a character-level language model, the alphabet of all possible characters is fixed. A one hot encoding is used both for learning input sequences and predicting output sequences. A one-to-one model is used where one step is predicted for each input time step. This means that input sequences may require specialized handling in order to be vectorized or formatted for efficiently training a supervised model. For example, given the sequence:

```
"hello world"
```

Listing 11.1: Example of a character sequence.

A dataset would need to be constructed such as:

```
'h' => 'e'
'e' => 'l'
'l' => 'l'
...
```

Listing 11.2: Example of a character sequence as a one-to-one model.

This could be presented as-is as a dataset of one time step samples, which could be quite limiting to the network (e.g. no BPTT). Alternately, it could be vectorized to a fixed-length input sequence for a many-to-one time step model, such as:

```
['h', 'e', 'l'] => 'l'
['e', 'l', 'l'] => 'o'
['l', 'l', 'o'] => ' '
...
```

Listing 11.3: Example of a character sequence as a many-to-one model.

Or, a fixed-length output sequence for a one-to-many time step model:

```
'h' => ['e', 'l', 'l']
'e' => ['l', 'l', 'o']
'l' => ['l', 'o', ' ']
...
```

Listing 11.4: Example of a character sequence as a one-to-many model.

Or some variation on these approaches. Note that the same vectorized representation would be required when making predictions, meaning that predicted characters would need to be presented as input for subsequent samples. This could be quite clumsy in implementation. The internal state of the network may need careful management, perhaps reset at choice locations in the input sequence (e.g. end of paragraph, page, or chapter) rather than at the end of each input sequence.

11.2 Shape Generation Problem

We can frame the problem of generating random shapes as a sequence generation problem. We can take drawing a rectangle as a sequence of points in the clockwise direction with 4 points in two-dimensional space:

- **Bottom Left** (BL): [0, 0]
- **Bottom Right** (BR): [1, 0]
- **Top Right** (TR): [1, 1]
- **Top Left**: [0, 1]

Each coordinate can be taken as one time step, with each of the `x` and `y` axes representing separate features. Starting from 0,0, the task is to draw the remaining 4 points of the rectangle with consistent widths and heights. We will frame this problem as a one-coordinate generation problem, e.g. a one-to-one sequence prediction problem. Given a coordinate, predict the next coordinate. Then given the coordinate predicted at the last time step, predict the next coordinate, and so on.

```
[0,0] => [x1, y1]
[x1,y1] => [x2, y2]
[x2,y2] => [x3, y3]
```

Listing 11.5: Example of coordinate prediction as a one-to-one model.

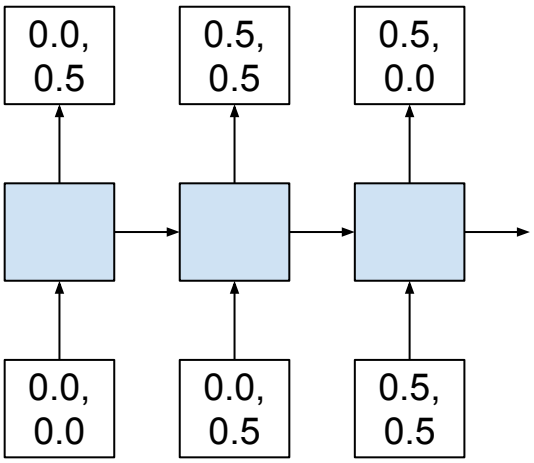

Figure 11.2: Shape generation problem framed with a one-to-one prediction model.

We can train the model by generating random rectangles and having the model predict the subsequent coordinates. Predicting the first coordinate from 0,0 will be impossible and predicting the height from the width will also be impossible, but we believe that given repeated exposure to whole rectangles that the model can learn how to draw new rectangles with consistent widths and heights. Implementing this in Python involves three steps:

1. Generate Random Rectangles
2. Plot Rectangles
3. Rectangle to Sequence

11.2.1 Generate Random Rectangles

We can use the `random()` function to generate random numbers between 0 and 1. For a given rectangle, we can use the `random()` function to define the width and height of the rectangle.

```
width, height = random(), random()
```

Listing 11.6: Example of generating a random width and height.

We can then use the width and height to define the 4 points of the rectangle, working in a clockwise direction, starting at `x=0,y=0`.

```
[0.0, 0.0]
[width, 0.0]
[width, height]
[0.0, height]
```

Listing 11.7: Example of coordinate sequence for a random rectangle.

The function below, named `random_rectangle()`, creates a random rectangle and returns the two-dimensional points as a list.

```
from random import random

# generate a rectangle with random width and height
def random_rectangle():
  width, height = random(), random()
  points = list()
  # bottom left
  points.append([0.0, 0.0])
  # bottom right
  points.append([width, 0.0])
  # top right
  points.append([width, height])
  # top left
  points.append([0.0, height])
  return points

rect = random_rectangle()
print(rect)
```

Listing 11.8: Example of generating a random rectangle.

Running the example creates one random rectangle and prints out the coordinates. The specific random rectangle you generate will differ.

```
[[0.0, 0.0],
 [0.40655549320386086, 0.0],
 [0.40655549320386086, 0.5040019272316 54],
 [0.0, 0.5040019272316 54]]
```

Listing 11.9: Example output from generating a random rectangle.

11.2.2 Plot Rectangles

We chose two-dimensional shape generation primarily because we can visualize the results. Therefore, we need a way of easily plotting a rectangle, either generated randomly or *predicted* by a model later. We can use the Matplotlib library to draw a path of coordinates.

This involves defining a `Path` for the shape that involves both a list of coordinates and the `Path` movements. The list of coordinates would be the list of coordinates we generated randomly with the addition of the first coordinate to the end of the list to close the polygon. The path is a series of movements, such as `MOVETO` to start the sequence, `LINETO` to connect points, and `CLOSEPOLY` to close the polygon.

```
# close the rectangle path
rect.append(rect[0])
# define path
codes = [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY]
path = Path(rect, codes)
```

Listing 11.10: Example of turning a rectangle coordinates into a shape.

We can define a `PathPatch` from the path and draw it directly on a Matplotlib plot.

```
axis = pyplot.gca()
patch = PathPatch(path)
```

```
# add shape to plot
axis.add_patch(patch)
```

Listing 11.11: Example of plotting a shape.

The function below named `plot_rectangle()` takes a list of 4 rectangle points as input and will plot the rectangle and show the plot. The axis bounds are adjusted to ensure the shape fits nicely in the 0-1 bounds.

```
# plot a rectangle
def plot_rectangle(rect):
  # close the rectangle path
  rect.append(rect[0])
  # define path
  codes = [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY]
  path = Path(rect, codes)
  axis = pyplot.gca()
  patch = PathPatch(path)
  # add shape to plot
  axis.add_patch(patch)
  axis.set_xlim(-0.1,1.1)
  axis.set_ylim(-0.1,1.1)
  pyplot.show()
```

Listing 11.12: Function to plot a rectangle.

Below is an example demonstrating this function plotting a randomly generated rectangle.

```
from random import random
from matplotlib import pyplot
from matplotlib.patches import PathPatch
from matplotlib.path import Path

# generate a rectangle with random width and height
def random_rectangle():
  width, height = random(), random()
  points = list()
  # bottom left
  points.append([0.0, 0.0])
  # bottom right
  points.append([width, 0.0])
  # top right
  points.append([width, height])
  # top left
  points.append([0.0, height])
  return points

# plot a rectangle
def plot_rectangle(rect):
  # close the rectangle path
  rect.append(rect[0])
  # define path
  codes = [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY]
  path = Path(rect, codes)
  axis = pyplot.gca()
  patch = PathPatch(path)
  # add shape to plot
```

```
  axis.add_patch(patch)
  axis.set_xlim(-0.1,1.1)
  axis.set_ylim(-0.1,1.1)
  pyplot.show()

rect = random_rectangle()
plot_rectangle(rect)
```

Listing 11.13: Example of generating and plotting a rectangle.

Running the example generates a random rectangle and plots it to the screen. The specific rectangle generated will vary each time the code is run.

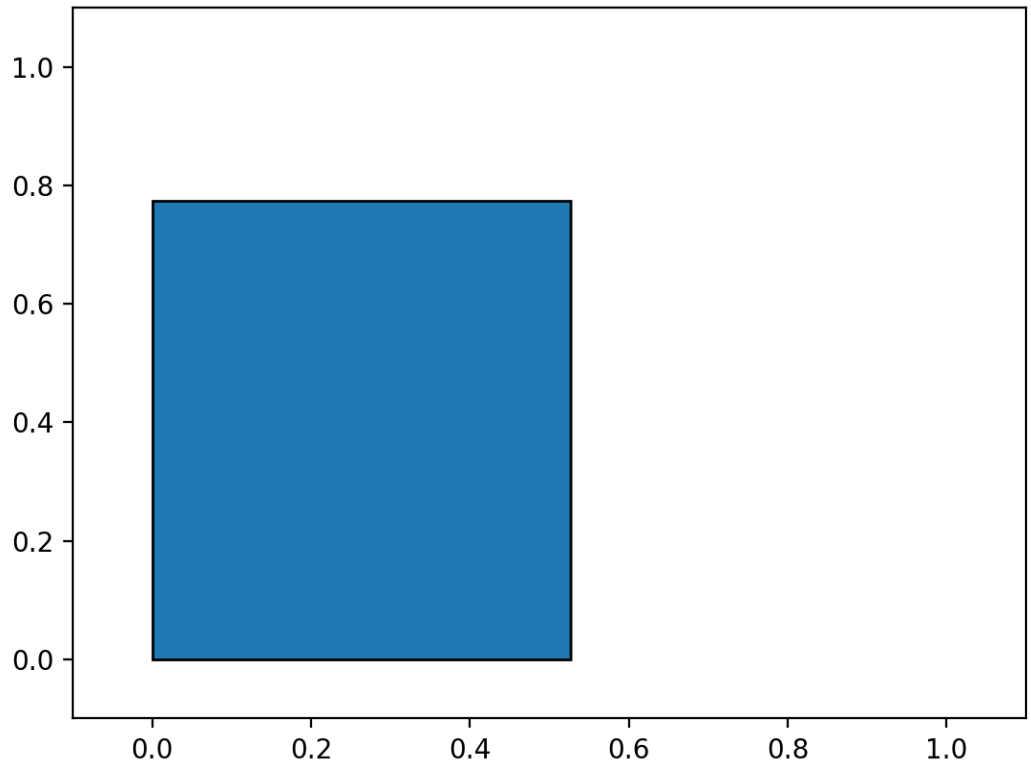

Figure 11.3: Plot of a random rectangle.

11.2.3 Rectangle to Sequence

Finally, we need to convert the sequence of 2D coordinates from a randomly generated rectangle into something we can use to train an LSTM model. Given a sequence of 4 points for a rectangle:

```
[BL, BR, TR, TL]
```

Listing 11.14: Example of a sequence of points in a rectangle.

We can create 3 samples with 1 time step and 1 feature as follows:

```
[BL] => [BR]
[BR] => [TR]
[TR] => [TL]
```

Listing 11.15: Example of a sequence of points for a one-to-one model.

The function below named `get_samples()` will generate a new random rectangle and convert it into a dataset of 3 samples.

```
# generate input and output sequences for one random rectangle
def get_samples():
  # generate rectangle
  rect = random_rectangle()
  X, y = list(), list()
  # create input output pairs for each coordinate
  for i in range(1, len(rect)):
    X.append(rect[i-1])
    y.append(rect[i])
  # convert input sequence shape to have 1 time step and 2 features
  X, y = array(X), array(y)
  X = X.reshape((X.shape[0], 1, 2))
  return X, y
```

Listing 11.16: Function generate a random rectangle and returns a sequence of points.

The example below demonstrates this function.

```
from random import random
from numpy import array

# generate a rectangle with random width and height
def random_rectangle():
  width, height = random(), random()
  points = list()
  # bottom left
  points.append([0.0, 0.0])
  # bottom right
  points.append([width, 0.0])
  # top right
  points.append([width, height])
  # top left
  points.append([0.0, height])
  return points

# generate input and output sequences for one random rectangle
def get_samples():
  # generate rectangle
  rect = random_rectangle()
  X, y = list(), list()
  # create input output pairs for each coordinate
  for i in range(1, len(rect)):
    X.append(rect[i-1])
    y.append(rect[i])
  # convert input sequence shape to have 1 time step and 2 features
  X, y = array(X), array(y)
  X = X.reshape((X.shape[0], 1, 2))
  return X, y
```

```
X, y = get_samples()
for i in range(X.shape[0]):
  print(X[i][0], '=>', y[i])
```

Listing 11.17: Example of generating a rectangle and preparing it for LSTM models.

Running this example shows the input and output coordinates for each sample. The specific coordinates will vary each time the example is run.

```
[ 0.  0.] => [ 0.07745734 0.        ]
[ 0.07745734 0.        ] => [ 0.07745734 0.86579881]
[ 0.07745734 0.86579881] => [ 0.         0.86579881]
```

Listing 11.18: Example output from generating a rectangle and preparing it for LSTM models.

11.3 Define and Compile the Model

We are now ready to define an LSTM model to address the shape prediction problem. The model will expect 1 time step input each with 2 features for the `x` and `y` axis of a single coordinate. We will use 10 memory cells in the `LSTM` hidden layer. A large capacity is not required for this problem and 10 cells were found with a little trial and error.

```
model = Sequential()
model.add(LSTM(10, input_shape=(1, 2)))
```

Listing 11.19: Example of adding the input layer.

The output of the network is a single coordinate comprised of `x` and `y` values. We will use a `Dense` layer with 2 neurons and a linear activation function.

```
model.add(Dense(2, activation='linear'))
```

Listing 11.20: Example of adding the output layer.

The model minimizes the mean absolute error (`mae`) loss function and the efficient Adam optimization algorithm is used to fit the network weights. The full model definition is listed below.

```
# define model
model = Sequential()
model.add(LSTM(10, input_shape=(1, 2)))
model.add(Dense(2, activation='linear'))
model.compile(loss='mae', optimizer='adam')
model.summary()
```

Listing 11.21: Example of defining and compiling the Generative LSTM.

Running the example prints a summary of the model structure. We can see that indeed the model is small.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 10)                520
_________________________________________________________________
```

```
dense_1 (Dense)          (None, 2)             22
=================================================================
Total params: 542
Trainable params: 542
Non-trainable params: 0
_________________________________________________________________
```

Listing 11.22: Example output from defining and compiling the Generative LSTM.

11.4 Fit the Model

The model can be fit by generating sequences one at a time and using them to update the model weights. Each sequence is comprised of 3 samples at the end of which the model weights will be updated and the internal state of each LSTM cell will be reset. This allows the model's memory to be focused on the specific input values of each different sequence. The number of samples generated represents a proxy for the number of training epochs; here we will use 25,000 randomly generated sequences of rectangle points. This configuration was found after a little trial and error. The order of the samples matters, therefore the shuffling of samples is turned off.

```
# fit model
for i in range(25000):
  X, y = get_samples()
  model.fit(X, y, epochs=1, verbose=2, shuffle=False)
```

Listing 11.23: Example of fitting a compiled Generative LSTM.

Fitting the model prints the loss at the end of each epoch.

```
...
Epoch 1/1
0s - loss: 0.0496
Epoch 1/1
0s - loss: 0.0757
Epoch 1/1
0s - loss: 0.0474
Epoch 1/1
0s - loss: 0.3612
Epoch 1/1
0s - loss: 0.0564
```

Listing 11.24: Example output from fitting a compiled Generative LSTM.

11.5 Make Predictions with the Model

Once the model is fit, we can use it to generate new rectangles. We can do this by defining the starting point of the rectangle as `[0,0]`, passing this as an input to get the next predicted coordinate. This defines the width of the rectangle.

```
# use [0,0] to seed the generation process
last = array([0.0,0.0]).reshape((1, 1, 2))
# predict the next coordinate
```

```
yhat = model.predict(last, verbose=0)
```

Listing 11.25: Example for predicting the rectangle width.

We can then use the predicted coordinate as an input to predict the next point. This will define the height of the rectangle.

```
# use this output as input for the next prediction
last = yhat.reshape((1, 1, 2))
# predict the next coordinate
yhat = model.predict(last, verbose=0)
```

Listing 11.26: Example for predicting the rectangle height.

The process is then repeated one more time to generate the remaining coordinate that is expected to remain consistent with the width and height already defined in the first three points. The function below named `generate_rectangle()` wraps up this functionality and generates a new rectangle using a fit model, then returns the list of points.

```
# use a fit LSTM model to generate a new rectangle from scratch
def generate_rectangle(model):
  rect = list()
  # use [0,0] to seed the generation process
  last = array([0.0,0.0]).reshape((1, 1, 2))
  rect.append([[y for y in x] for x in last[0]][0])
  # generate the remaining 3 coordinates
  for i in range(3):
    # predict the next coordinate
    yhat = model.predict(last, verbose=0)
    # use this output as input for the next prediction
    last = yhat.reshape((1, 1, 2))
    # store coordinate
    rect.append([[y for y in x] for x in last[0]][0])
  return rect
```

Listing 11.27: Function for generating a rectangle from scratch.

We can use this function to make a prediction, then use the previously defined function `plot_rectangle()` to plot the results.

```
# generate new shapes from scratch
rect = generate_rectangle(model)
plot_rectangle(rect)
```

Listing 11.28: Example of generating a rectangle from scratch and plotting it.

Running this example generates one rectangle and plots it to the screen. We can see that indeed the rectangle does a reasonable job of maintaining the consistent proportions of width and height. The specific rectangle generated will vary each time the code is run.

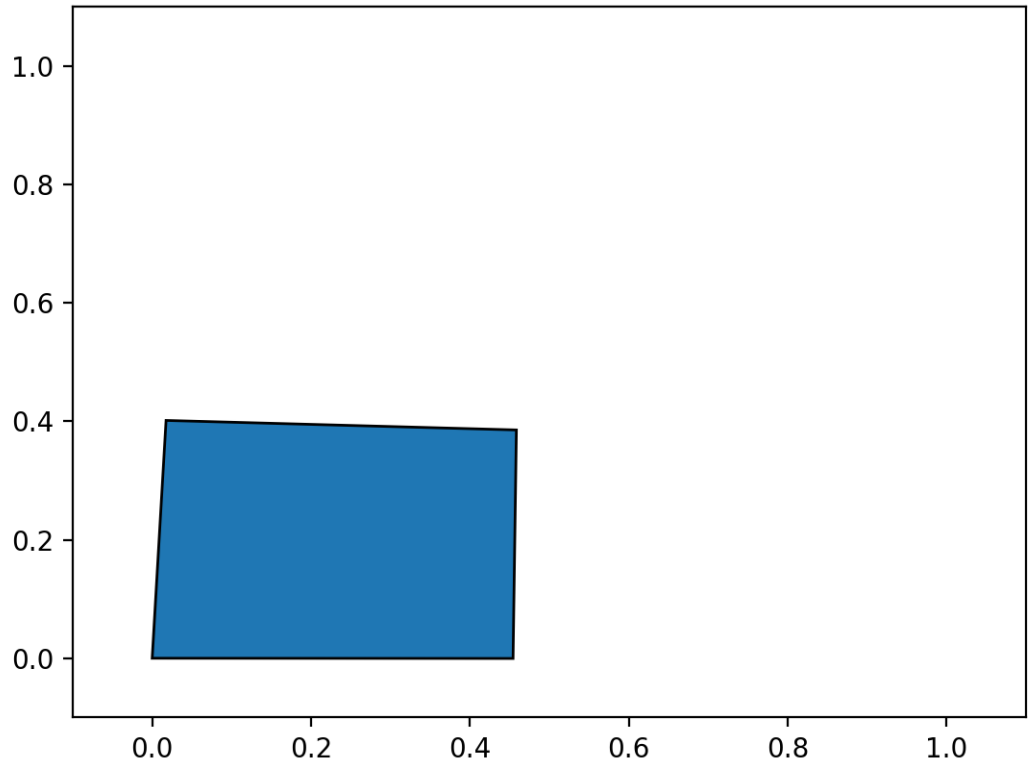

Figure 11.4: Plot of a rectangle generated by the Generative LSTM model.

11.6 Evaluate the Model

It is difficult to evaluate a generative model. In this case, we have a well defined expectation on the generated shape (rectangular) that can be evaluated quantitatively and qualitatively. In many problems, this may not be the case and you may have to rely on qualitative evaluation.

We can achieve a qualitative evaluation of rectangles via visualization. Below are a few examples of rectangles generated after different amounts of training to show the qualitative improvement in model skill as the number of training examples is increased.

11.6.1 100 Training Examples

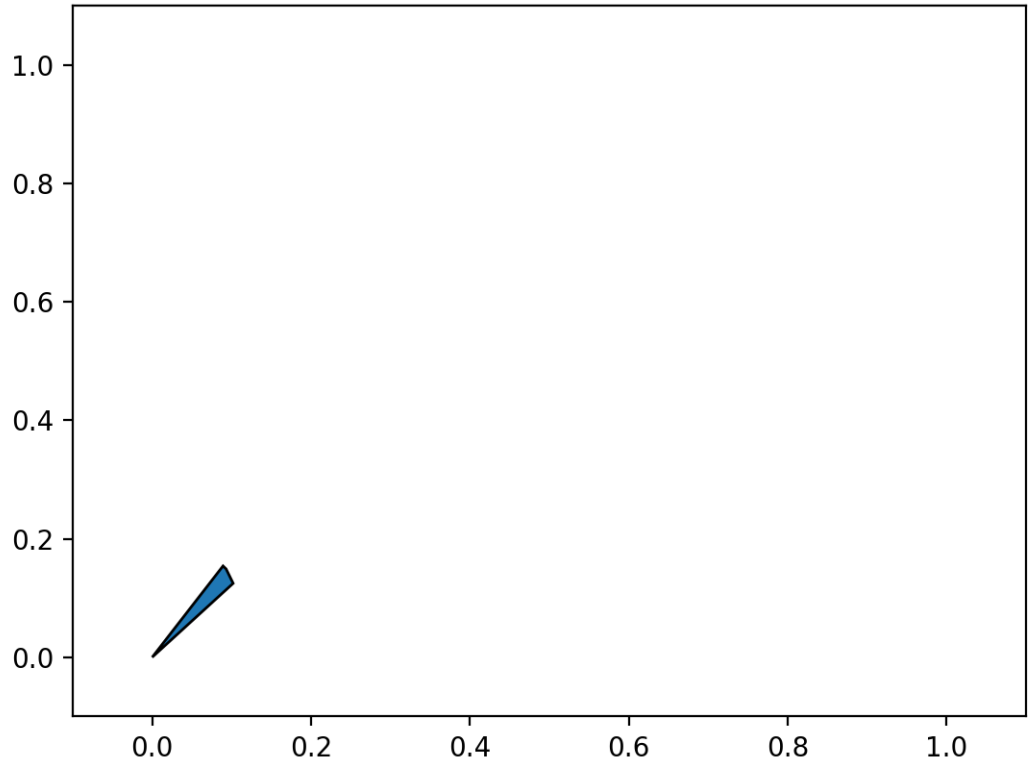

Figure 11.5: Plot of a rectangle generated by the Generative LSTM model after 100 examples.

11.6.2 500 Training Examples

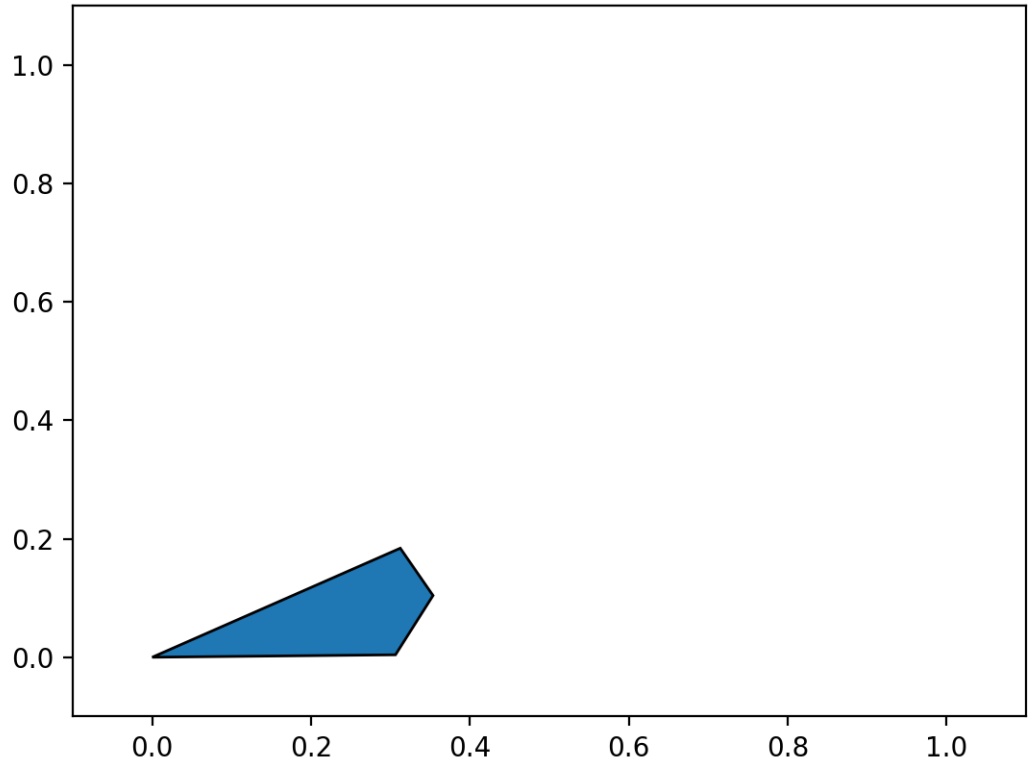

Figure 11.6: Plot of a rectangle generated by the Generative LSTM model after 500 examples.

11.6.3 1,000 Training Examples

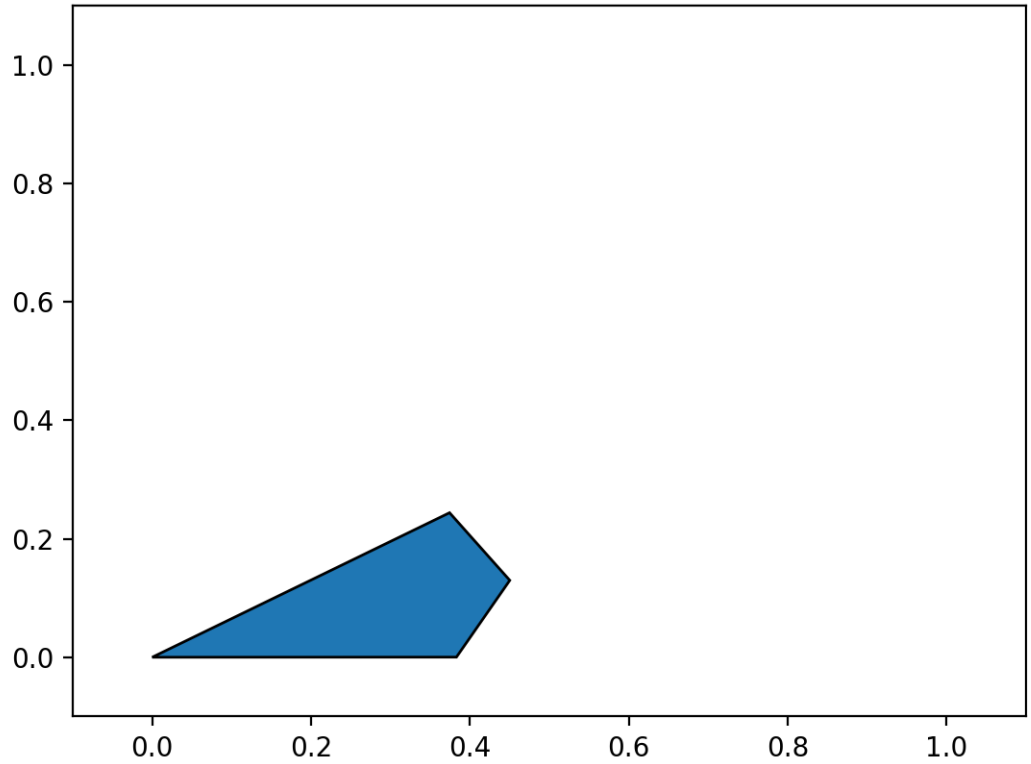

Figure 11.7: Plot of a rectangle generated by the Generative LSTM model after 1000 examples.

11.6.4 5,000 Training Examples

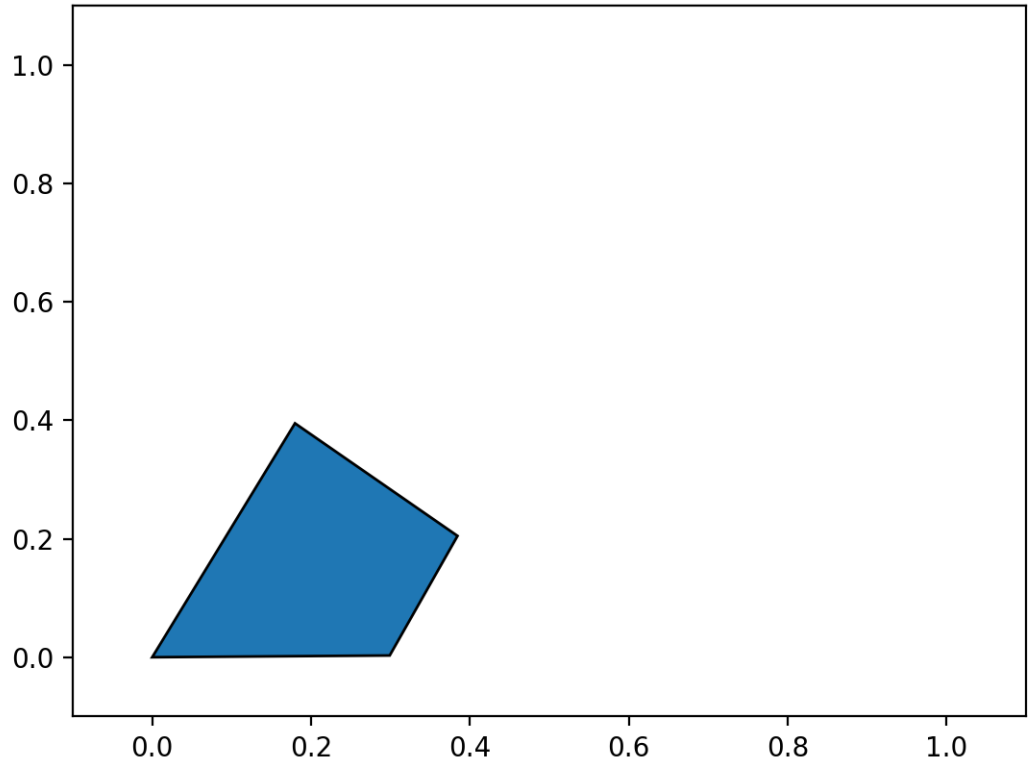

Figure 11.8: Plot of a rectangle generated by the Generative LSTM model after 5000 examples.

11.6.5 10,000 Training Examples

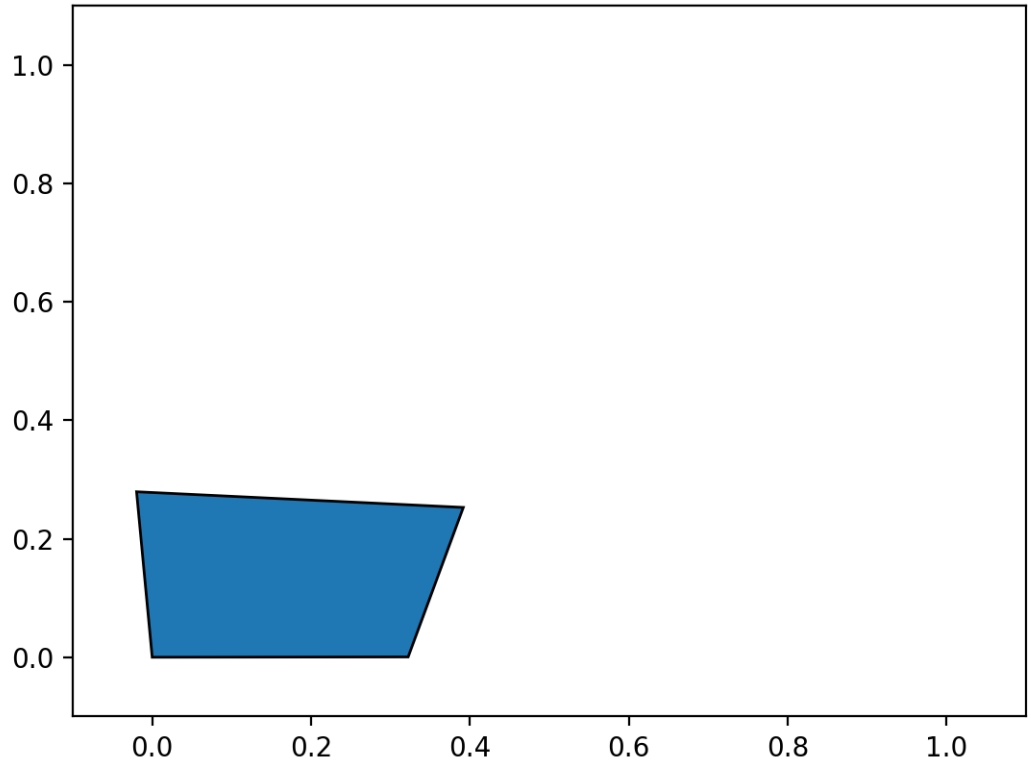

Figure 11.9: Plot of a rectangle generated by the Generative LSTM model after 10000 examples.

11.7 Complete Example

The complete code listing is provided below for your reference.

```
from random import random
from numpy import array
from matplotlib import pyplot
from matplotlib.patches import PathPatch
from matplotlib.path import Path
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# generate a rectangle with random width and height
def random_rectangle():
  width, height = random(), random()
  points = list()
  # bottom left
  points.append([0.0, 0.0])
  # bottom right
```

```
  points.append([width, 0.0])
  # top right
  points.append([width, height])
  # top left
  points.append([0.0, height])
  return points

# plot a rectangle
def plot_rectangle(rect):
  # close the rectangle path
  rect.append(rect[0])
  # define path
  codes = [Path.MOVETO, Path.LINETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY]
  path = Path(rect, codes)
  axis = pyplot.gca()
  patch = PathPatch(path)
  # add shape to plot
  axis.add_patch(patch)
  axis.set_xlim(-0.1,1.1)
  axis.set_ylim(-0.1,1.1)
  pyplot.show()

# generate input and output sequences for one random rectangle
def get_samples():
  # generate rectangle
  rect = random_rectangle()
  X, y = list(), list()
  # create input output pairs for each coordinate
  for i in range(1, len(rect)):
    X.append(rect[i-1])
    y.append(rect[i])
  # convert input sequence shape to have 1 time step and 2 features
  X, y = array(X), array(y)
  X = X.reshape((X.shape[0], 1, 2))
  return X, y

# use a fit LSTM model to generate a new rectangle from scratch
def generate_rectangle(model):
  rect = list()
  # use [0,0] to seed the generation process
  last = array([0.0,0.0]).reshape((1, 1, 2))
  rect.append([[y for y in x] for x in last[0]][0])
  # generate the remaining 3 coordinates
  for _ in range(3):
    # predict the next coordinate
    yhat = model.predict(last, verbose=0)
    # use this output as input for the next prediction
    last = yhat.reshape((1, 1, 2))
    # store coordinate
    rect.append([[y for y in x] for x in last[0]][0])
  return rect

# define model
model = Sequential()
model.add(LSTM(10, input_shape=(1, 2)))
model.add(Dense(2, activation='linear'))
```

```
model.compile(loss='mae', optimizer='adam')
model.summary()

# fit model
for i in range(25000):
  X, y = get_samples()
  model.fit(X, y, epochs=1, verbose=2, shuffle=False)

# generate new shapes from scratch
rect = generate_rectangle(model)
plot_rectangle(rect)
```

Listing 11.29: Example of the Generative LSTM on the Rectangle Generation problem.

11.8 Further Reading

This section provides some resources for further reading.

11.8.1 Research Papers

- *Generating Text with Recurrent Neural Networks*, 2011.
- *Generating Sequences With Recurrent Neural Networks*, 2013.
  https://arxiv.org/abs/1308.0850
- *TTS Synthesis with Bidirectional LSTM based Recurrent Neural Networks*, 2014.
- *A First Look at Music Composition using LSTM Recurrent Neural Networks*, 2002.
- *Jazz Melody Generation from Recurrent Network Learning of Several Human Melodies*, 2005.

11.8.2 Articles

- *The Unreasonable Effectiveness of Recurrent Neural Networks*, 2015.
  http://karpathy.github.io/2015/05/21/rnn-effectiveness/

11.8.3 APIs

- Python `random` API.
  https://docs.python.org/3/library/random.html
- Matplotlib `Path` API.
  https://matplotlib.org/api/path_api.html
- Matplotlib `Patch` API.
  https://matplotlib.org/api/patches_api.html

11.9 Extensions

Do you want to dive deeper into Generative LSTMs? This section lists some challenging extensions to this lesson.

- List 5 other examples of problems, other than language modeling, where Generative LSTMs could be used.
- Design and execute an experiment to compare model size (number of cells) to qualitative model skill (plots).
- Update the example so that random rectangles are comprised of more points (e.g. points on the mid-width and mid-height of the rectangle) then tune the LSTM to achieve good skill.
- Develop a function to estimate rectangle error in terms of inconsistent width and height and use as a loss function and metric when fitting the Generative LSTMs.
- Update the example to learn to generate a different shape, such as a circle, star, or cross.

Post your extensions online and share the link with me. I'd love to see what you come up with!

11.10 Summary

In this lesson, you discovered how to develop a Generative LSTM model. Specifically, you learned:

- How LSTMs can be used as sequence generation models.
- How to frame shape drawing as a sequence generation problem.
- How to develop an LSTM to generate shapes.

In the next lesson, you will discover how to get the most out of your LSTM models.

Part IV

Advanced

Chapter 12

How to Diagnose and Tune LSTMs

12.0.1 Lesson Goal

The goal of this lesson is to learn how to tune LSTM hyperparameters. After completing this lesson, you will know:

- How to develop a robust evaluation of the skill of your LSTM models.
- How to use learning curves to diagnose the behavior of your LSTM models.
- How to tune the problem framing, structure, and learning behavior of your LSTM models.

12.0.2 Lesson Overview

This lesson is divided into 5 parts; they are:

1. Evaluating LSTM Models Robustly.
2. Diagnosing Underfitting and Overfitting.
3. Tune Problem Framing.
4. Tune Model Structure.
5. Tune Learning Behavior.

Let's get started.

12.1 Evaluating LSTM Models Robustly

In this section, you will discover the procedure to use to develop a robust estimate of the skill of your LSTM models on unseen data.

12.1.1 The Beginner's Mistake

You fit the model to your training data and evaluate it on the test dataset, then report the skill. Perhaps you use k-fold cross-validation to evaluate the model, then report the skill of the model. This is a mistake made by beginners.

It looks like you're doing the right thing, but there is a key issue you have not accounted for: *deep learning models are stochastic*. Artificial neural networks like LSTMs use randomness while being fit on a dataset, such as random initial weights and random shuffling of data during each training epoch during stochastic gradient descent. This means that each time the same model is fit on the same data, it may give different predictions and in turn have different overall skill.

12.1.2 Estimating Model Skill

We don't have all possible data; if we did, we would not need to make predictions. We have a limited sample of data, and from it we need to discover the best model we can.

We do that by splitting the data into two parts, fitting a model or specific model configuration on the first part of the data and using the fit model to make predictions on the rest, then evaluating the skill of those predictions. This is called a train-test split and we use the skill as an estimate for how well we think the model will perform in practice when it makes predictions on new data. For example, here's some pseudocode for evaluating a model using a train-test split:

```
train, test = random_split(data)
model = fit(train.X, train.y)
predictions = model.predict(test.X)
skill = compare(test.y, predictions)
```

Listing 12.1: Pseudocode for evaluating model skill.

A train-test split is a good approach to use if you have a lot of data or a very slow model to train, but the resulting skill score for the model will be noisy because of the randomness in the data (variance of the model). This means that the same model fit on different data will give different model skill scores. If we have the resources, we would use k-fold cross-validation. But this is generally not possible given the use of large dataset in deep learning and the slow speed of training the model.

12.1.3 Estimating a Stochastic Model's Skill

Stochastic models, like deep neural networks, add an additional source of randomness. This additional randomness gives the model more flexibility when learning, but can make the model less stable (e.g. different results when the same model is trained on the same data). This is different from model variance that gives different results when the same model is trained on different data.

To get a robust estimate of the skill of a stochastic model, we must take this additional source of variance into account; we must control for it. A robust approach is to repeat the experiment of evaluating a stochastic model multiple times. For example:

```
scores = list()
for i in repeats:
  train, test = random_split(data)
```

```
  model = fit(train.X, train.y)
  predictions = model.predict(test.X)
  skill = compare(test.y, predictions)
  scores.append(skill)
final_skill = mean(scores)
```

Listing 12.2: Pseudocode for evaluating stochastic model skill.

This is my recommended procedure for estimating the skill of a deep learning model.

12.1.4 How Unstable Are Neural Networks?

It depends on your problem, on the network, and on its configuration. I would recommend performing a sensitivity analysis to find out. Evaluate the same model on the same data many times (30, 100, or thousands) and only vary the seed for the random number generator. Then review the mean and standard deviation of the skill scores produced. The standard deviation (average distance of scores from the mean score) will give you an idea of just how unstable your model is.

12.1.5 How Many Repeats?

I would recommend at least 30, perhaps 100, even thousands, limited only by your time and computer resources, and diminishing returns (e.g. standard error on the mean skill). More rigorously, I would recommend an experiment that looked at the impact on estimated model skill versus the number of repeats and the calculation of the standard error (how much the mean estimated performance differs from the true underlying population mean).

12.1.6 After Evaluating Modes

Evaluating your models is a means to an end. It helps you choose which model to use and which hyperparameters to use to configure it. Once you have chosen a model, you must finalize it. This involves fitting the model on all available data and saving it for later use in making predictions on new data where we don't know the actual outcome. We cover the process of model finalization in more detail in a later chapter.

12.2 Diagnosing Underfitting and Overfitting

In this section, you will discover how to use plots of the learning curves of LSTM models during training to diagnose overfitting and underfitting.

12.2.1 Training History in Keras

You can learn a lot about the behavior of your model by reviewing its performance over time. LSTM models are trained by calling the `fit()` function. This function returns a variable called history that contains a trace of the loss and any other metrics specified during the compilation of the model. These scores are recorded at the end of each epoch.

```
...
history = model.fit(...)
```

Listing 12.3: Example of assigning history after fitting an LSTM model.

For example, if your model was compiled to optimize the log loss (`binary_crossentropy`) and measure accuracy each epoch, then the log loss and accuracy will be calculated and recorded in the history trace for each training epoch. Each score is accessed by a key in the history object returned from calling `fit()`. By default, the loss optimized when fitting the model is called `loss` and accuracy is called `acc`.

```
...
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X, Y, epochs=100)
print(history.history['loss'])
print(history.history['accuracy'])
```

Listing 12.4: Example of printing history after fitting an LSTM model.

Keras also allows you to specify a separate validation dataset while fitting your model that can also be evaluated using the same loss and metrics. This can be done by setting the `validation_split` argument on `fit()` to use a portion of the training data as a validation dataset (specifically a ration between 0.0 and 1.0).

```
...
history = model.fit(X, Y, epochs=100, validation_split=0.33)
```

Listing 12.5: Example history that validation loss calculated on a subset of the training data.

This can also be done by setting the `validation_data` argument and passing a tuple of `X` and `y` datasets.

```
...
history = model.fit(X, Y, epochs=100, validation_data=(valX, valY))
```

Listing 12.6: Example history that contains validation loss on a new dataset.

The metrics evaluated on the validation dataset are keyed using the same names, with a `val_` prefix.

```
...
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X, Y, epochs=100, validation_split=0.33)
print(history.history['loss'])accuracy
print(history.history['accuracy'])
print(history.history['val_loss'])
print(history.history['val_accuracy'])
```

Listing 12.7: Example of printing train and validation loss and accuracy.

12.2.2 Diagnostic Plots

The training history of your LSTM models can be used to diagnose the behavior of your model. You can plot the performance of your model using the Matplotlib library. For example, you can plot training loss vs test loss as follows:

```
from matplotlib import pyplot
...
history = model.fit(X, Y, epochs=100, validation_data=(valX, valY))
pyplot.plot(history.history['loss'])
pyplot.plot(history.history['val_loss'])
pyplot.title('model train vs validation loss')
pyplot.ylabel('loss')
pyplot.xlabel('epoch')
pyplot.legend(['train', 'validation'], loc='upper right')
pyplot.show()
```

Listing 12.8: Example of plotting train and validation loss and accuracy.

Creating and reviewing these plots can help to inform you about possible new configurations to try in order to get better performance from your model. Next we will look at some examples. We will consider model skill on the train and validation sets in terms of loss that is minimized. You can use any metric that is meaningful on your problem.

12.2.3 Underfit

An underfit model is one that is demonstrated to perform well on the training dataset and poor on the test dataset. This can be diagnosed from a plot where the training loss is lower than the validation loss, and the validation loss has a trend that suggests further improvements are possible. A small contrived example of an underfit LSTM model is provided below.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot
from numpy import array

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# return validation data
def get_val():
  seq = [[0.5, 0.6], [0.6, 0.7], [0.7, 0.8], [0.8, 0.9], [0.9, 1.0]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# define model
model = Sequential()
model.add(LSTM(10, input_shape=(1,1)))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mse', optimizer='adam')
# fit model
```

```
X,y = get_train()
valX, valY = get_val()
history = model.fit(X, y, epochs=100, validation_data=(valX, valY), shuffle=False)
# plot train and validation loss
pyplot.plot(history.history['loss'])
pyplot.plot(history.history['val_loss'])
pyplot.title('model train vs validation loss')
pyplot.ylabel('loss')
pyplot.xlabel('epoch')
pyplot.legend(['train', 'validation'], loc='upper right')
pyplot.show()
```

Listing 12.9: Example of an underfit LSTM plotting train and validation loss.

Running this example produces a plot of train and validation loss showing the characteristic of an underfit model. In this case, performance may be improved by increasing the number of training epochs.

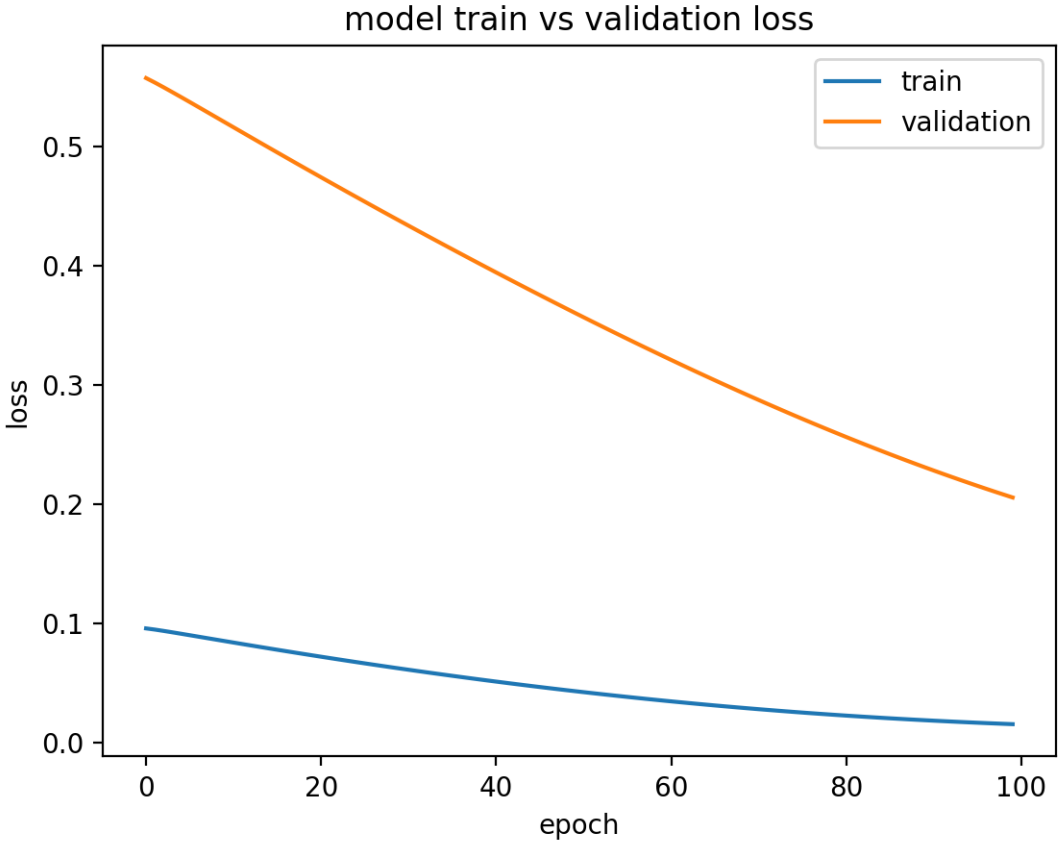

Figure 12.1: Diagnostic line plot of underfit model with more training to do.

Alternately, a model may be underfit if performance on the training set is better than the validation set and performance has leveled off. Below is an example of an underfit model with insufficient memory cells.

```
from keras.models import Sequential
```

```
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot
from numpy import array

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((5, 1, 1))
  return X, y

# return validation data
def get_val():
  seq = [[0.5, 0.6], [0.6, 0.7], [0.7, 0.8], [0.8, 0.9], [0.9, 1.0]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# define model
model = Sequential()
model.add(LSTM(1, input_shape=(1,1)))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mae', optimizer='sgd')
# fit model
X,y = get_train()
valX, valY = get_val()
history = model.fit(X, y, epochs=300, validation_data=(valX, valY), shuffle=False)
# plot train and validation loss
pyplot.plot(history.history['loss'])
pyplot.plot(history.history['val_loss'])
pyplot.title('model train vs validation loss')
pyplot.ylabel('loss')
pyplot.xlabel('epoch')
pyplot.legend(['train', 'validation'], loc='upper right')
pyplot.show()
```

Listing 12.10: Example of an underfit LSTM plotting train and validation loss.

Running this example shows the characteristic of an underfit model that appears under-provisioned. In this case, performance may be improved by increasing the capacity of the model, such as the number of memory cells in a hidden layer or number of hidden layers.

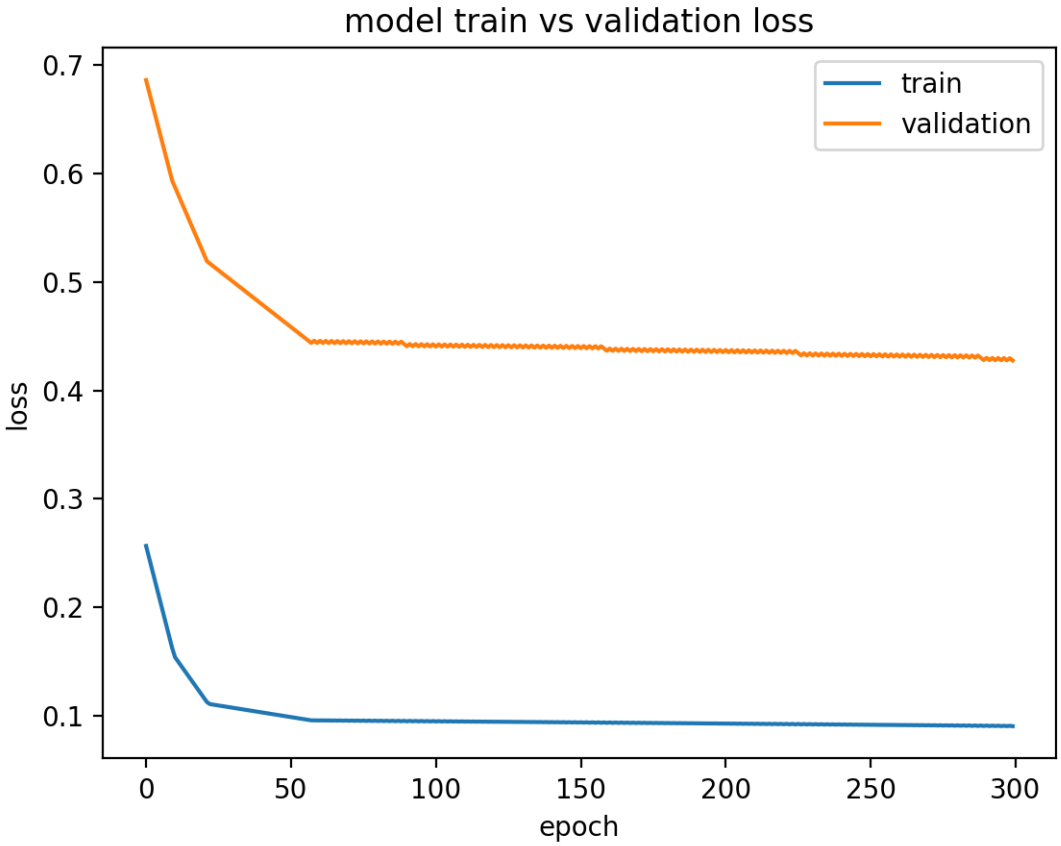

Figure 12.2: Diagnostic line plot of underfit model where are larger model may be needed.

12.2.4 Good Fit

A good fit is a case where the performance of the model is good on both the train and validation sets. This can be diagnosed from a plot where the train and validation loss decrease and stabilize around the same point. The small example below demonstrates an LSTM model with a good fit.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot
from numpy import array

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((5, 1, 1))
  return X, y

# return validation data
def get_val():
  seq = [[0.5, 0.6], [0.6, 0.7], [0.7, 0.8], [0.8, 0.9], [0.9, 1.0]]
```

```
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# define model
model = Sequential()
model.add(LSTM(10, input_shape=(1,1)))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mse', optimizer='adam')
# fit model
X,y = get_train()
valX, valY = get_val()
history = model.fit(X, y, epochs=800, validation_data=(valX, valY), shuffle=False)
# plot train and validation loss
pyplot.plot(history.history['loss'])
pyplot.plot(history.history['val_loss'])
pyplot.title('model train vs validation loss')
pyplot.ylabel('loss')
pyplot.xlabel('epoch')
pyplot.legend(['train', 'validation'], loc='upper right')
pyplot.show()
```

Listing 12.11: Example of a good fit LSTM plotting train and validation loss.

Running the example creates a line plot showing the train and validation loss meeting. Ideally, we would like to see model performance like this if possible, although this may not be possible on challenging problems with a lot of data.

Figure 12.3: Diagnostic line plot of well fit model.

12.2.5 Overfit

An overfit model is one where performance on the train set is good and continues to improve, whereas performance on the validation set improves to a point and then begins to degrade. This can be diagnosed from a plot where the train loss slopes down and the validation loss slopes down, hits an inflection point, and starts to slope up again. The example below demonstrates an overfit LSTM model.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot
from numpy import array

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((5, 1, 1))
  return X, y

# return validation data
```

```
def get_val():
  seq = [[0.5, 0.6], [0.6, 0.7], [0.7, 0.8], [0.8, 0.9], [0.9, 1.0]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# define model
model = Sequential()
model.add(LSTM(10, input_shape=(1,1)))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mse', optimizer='adam')
# fit model
X,y = get_train()
valX, valY = get_val()
history = model.fit(X, y, epochs=1200, validation_data=(valX, valY), shuffle=False)
# plot train and validation loss
pyplot.plot(history.history['loss'][500:])
pyplot.plot(history.history['val_loss'][500:])
pyplot.title('model train vs validation loss')
pyplot.ylabel('loss')
pyplot.xlabel('epoch')
pyplot.legend(['train', 'validation'], loc='upper right')
pyplot.show()
```

Listing 12.12: Example of an overfit LSTM plotting train and validation loss.

Running this example creates a plot showing the characteristic inflection point in validation loss of an overfit model. This may be a sign of too many training epochs. In this case, the model training could be stopped at the inflection point. Alternately, the number of training examples could be increased.

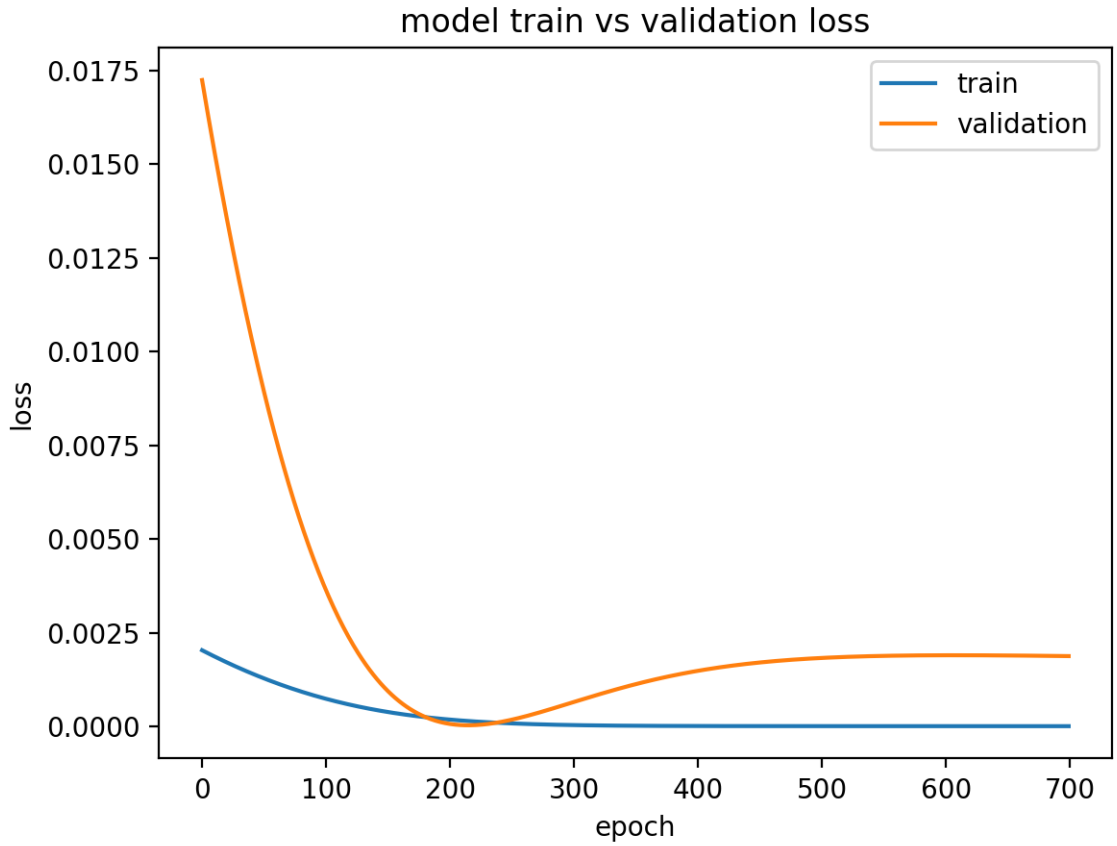

Figure 12.4: Diagnostic line plot of an overfit model.

12.2.6 Multiple Runs

LSTMs are stochastic, meaning that you will get a different diagnostic plot each run. It can be useful to repeat the diagnostic run multiple times (e.g. 5, 10, or 30). The train and validation traces from each run can then be plotted to give a more robust idea of the behavior of the model over time. The example below runs the same experiment a number of times before plotting the trace of train and validation loss for each run.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot
from numpy import array
from pandas import DataFrame

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((5, 1, 1))
  return X, y
```

```
# return validation data
def get_val():
  seq = [[0.5, 0.6], [0.6, 0.7], [0.7, 0.8], [0.8, 0.9], [0.9, 1.0]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# collect data across multiple repeats
train = DataFrame()
val = DataFrame()
for i in range(5):
  # define model
  model = Sequential()
  model.add(LSTM(10, input_shape=(1,1)))
  model.add(Dense(1, activation='linear'))
  # compile model
  model.compile(loss='mse', optimizer='adam')
  X,y = get_train()
  valX, valY = get_val()
  # fit model
  history = model.fit(X, y, epochs=300, validation_data=(valX, valY), shuffle=False)
  # story history
  train[str(i)] = history.history['loss']
  val[str(i)] = history.history['val_loss']

# plot train and validation loss across multiple runs
pyplot.plot(train, color='blue', label='train')
pyplot.plot(val, color='orange', label='validation')
pyplot.title('model train vs validation loss')
pyplot.ylabel('loss')
pyplot.xlabel('epoch')
pyplot.show()
```

Listing 12.13: Example of multiple diagnostic plots of train and validation loss.

In the resulting plot we can see that the general trend of underfitting holds across 5 runs and is a stronger case for perhaps increasing the number of training epochs.

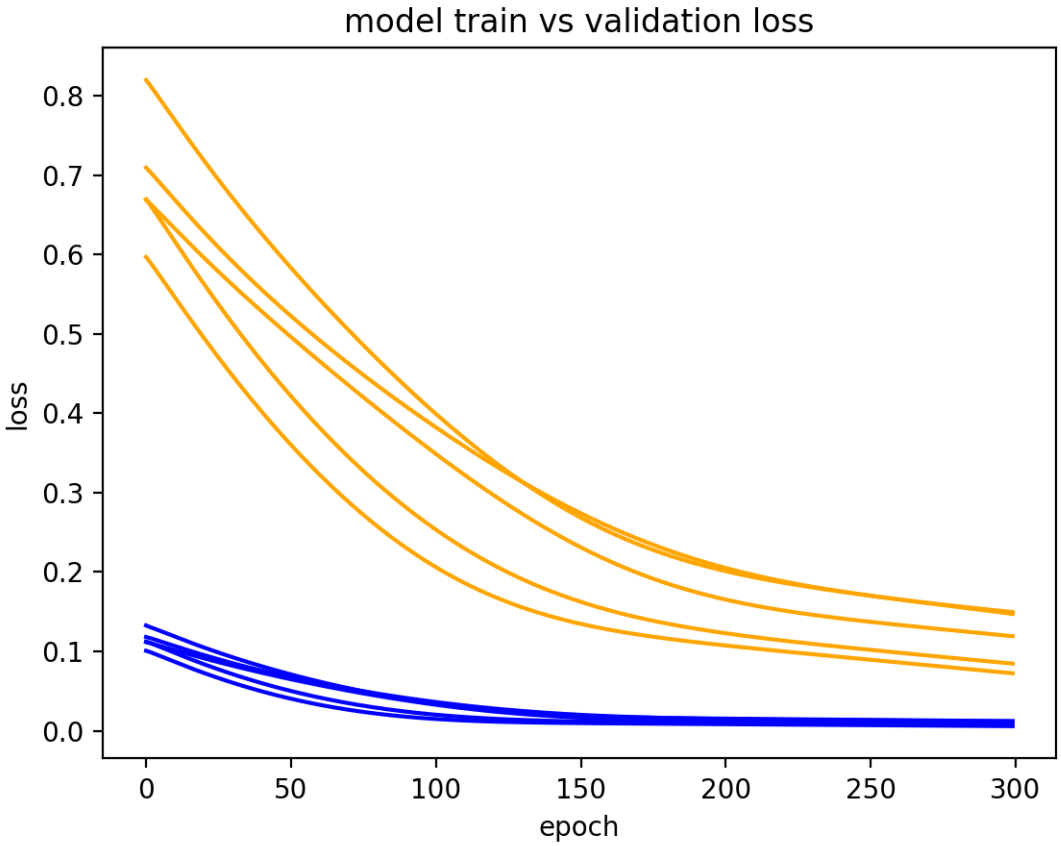

Figure 12.5: Diagnostic line plot of multiple model runs.

12.3 Tune Problem Framing

This section outlines areas of biggest leverage to consider when tuning the framing of your sequence prediction problem.

12.3.1 Value Scaling

Evaluate the effect of different data value scaling schemes on the skill of your model. Remember to update the activation function on the first hidden layer and/or the output layer to handle the range of values provided as input or predicted as output. Some schemes to try include:

- Normalize values.
- Standardize values.

12.3.2 Value Encoding

Evaluate the effect of different value encodings on the skill of your model. Sequences of labels, like characters or words, are often integer encoded and one hot encoded. Test the assumption

that this is the most skillful approach on your sequence prediction problem. Some encoding schemes to try include:

- Real-value encoding.
- Integer encoding.
- One hot encoding.

12.3.3 Stationarity

When working with a sequence of real values, like a time series, consider making the series stationary.

- **Remove Trends**: If the series contains variance in the mean (e.g. a trend), you can use differencing.
- **Remove Seasonality**: If the series contains a periodic cycle (e.g. seasonality), you can use seasonal adjustment.
- **Remove Variance**: If the series contains an increasing or decreasing variance, you can use a log or Box-Cox transform.

12.3.4 Input Sequence Length

The choice of the input sequence length may be specific to your problem domain, although sometimes it may be your choice as part of framing the problem for the LSTM. Evaluate the effect of using different input sequence lengths on the skill of your model. Remember, the length of the input sequence also impacts the Backpropagation through time used to estimate the error gradient when updating the weights. It can have an effect on how quickly the model learns and what is learned.

12.3.5 Sequence Model Type

There are 4 main sequence model types for a given sequence prediction problem:

- One-to-one.
- One-to-many.
- Many-to-one.
- Many-to-many.

Keras supports them all. Frame your problem using each of the sequence model types and evaluate model skill to help you choose a framing for your problem.

12.4 Tune Model Structure

This section outlines some areas of biggest leverage when tuning the structure of your LSTM Model.

12.4.1 Architecture

As we have seen, there are many LSTM architectures to choose from. Some architectures lend themselves to certain sequence prediction problems, although most are flexible enough that they may be adapted to your sequence prediction problems. Test your assumptions of architecture suitability. Evaluate the skill of each LSTM architecture listed in this book (and perhaps beyond) on your sequence prediction problem.

12.4.2 Memory Cells

We cannot know the *best* number of memory cells for a given sequence prediction problem or LSTM architecture. You must test a suite of different memory cells in your LSTM hidden layers to see what works best.

- Try grid searching the number of memory cells by 100s, 10s, or finer.
- Try using numbers of cells quoted in research papers.
- Try randomly searching the number of cells between 1 and 1000.

I often see round numbers of memory cells used such as 100 or 1000. I expect these were chosen on a whim. Below is a small example of grid searching the number of memory cells 1, 5 or 10 in the first hidden LSTM layer with a small number of repeats (5). You could use this example as a template for your own experiments.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot
from pandas import DataFrame
from numpy import array

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((5, 1, 1))
  return X, y

# return validation data
def get_val():
  seq = [[0.5, 0.6], [0.6, 0.7], [0.7, 0.8], [0.8, 0.9], [0.9, 1.0]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y
```

```
# fit an LSTM model
def fit_model(n_cells):
  # define model
  model = Sequential()
  model.add(LSTM(n_cells, input_shape=(1,1)))
  model.add(Dense(1, activation='linear'))
  # compile model
  model.compile(loss='mse', optimizer='adam')
  # fit model
  X,y = get_train()
  model.fit(X, y, epochs=500, shuffle=False, verbose=0)
  # evaluate model
  valX, valY = get_val()
  loss = model.evaluate(valX, valY, verbose=0)
  return loss

# define scope of search
params = [1, 5, 10]
n_repeats = 5
# grid search parameter values
scores = DataFrame()
for value in params:
  # repeat each experiment multiple times
  loss_values = list()
  for i in range(n_repeats):
    loss = fit_model(value)
    loss_values.append(loss)
    print('>%d/%d param=%f, loss=%f' % (i+1, n_repeats, value, loss))
  # store results for this parameter
  scores[str(value)] = loss_values
# summary statistics of results
print(scores.describe())
# box and whisker plot of results
scores.boxplot()
pyplot.show()
```

Listing 12.14: Example of grid searching the number of memory cells.

Running the example prints the progress of the search each iteration. Summary statistics of the results for each number of memory cells are then shown at the end.

```
>1/5 param=1.000000, loss=0.318098
>2/5 param=1.000000, loss=0.209199
>3/5 param=1.000000, loss=0.121033
>4/5 param=1.000000, loss=0.180944
>5/5 param=1.000000, loss=0.202586
>1/5 param=5.000000, loss=0.078900
>2/5 param=5.000000, loss=0.085735
>3/5 param=5.000000, loss=0.040546
>4/5 param=5.000000, loss=0.107958
>5/5 param=5.000000, loss=0.166691
>1/5 param=10.000000, loss=0.021442
>2/5 param=10.000000, loss=0.038652
>3/5 param=10.000000, loss=0.033825
>4/5 param=10.000000, loss=0.020847
>5/5 param=10.000000, loss=0.023117
```

```
             1         5         10
count  5.000000  5.000000  5.000000
mean   0.206372  0.095966  0.027577
std    0.071474  0.046404  0.008132
min    0.121033  0.040546  0.020847
25%    0.180944  0.078900  0.021442
50%    0.202586  0.085735  0.023117
75%    0.209199  0.107958  0.033825
max    0.318098  0.166691  0.038652
```

Listing 12.15: Example output of grid searching the number of memory cells.

A box and whisker plot of the final results is created to compare the distribution of model skill for each of the different model configurations.

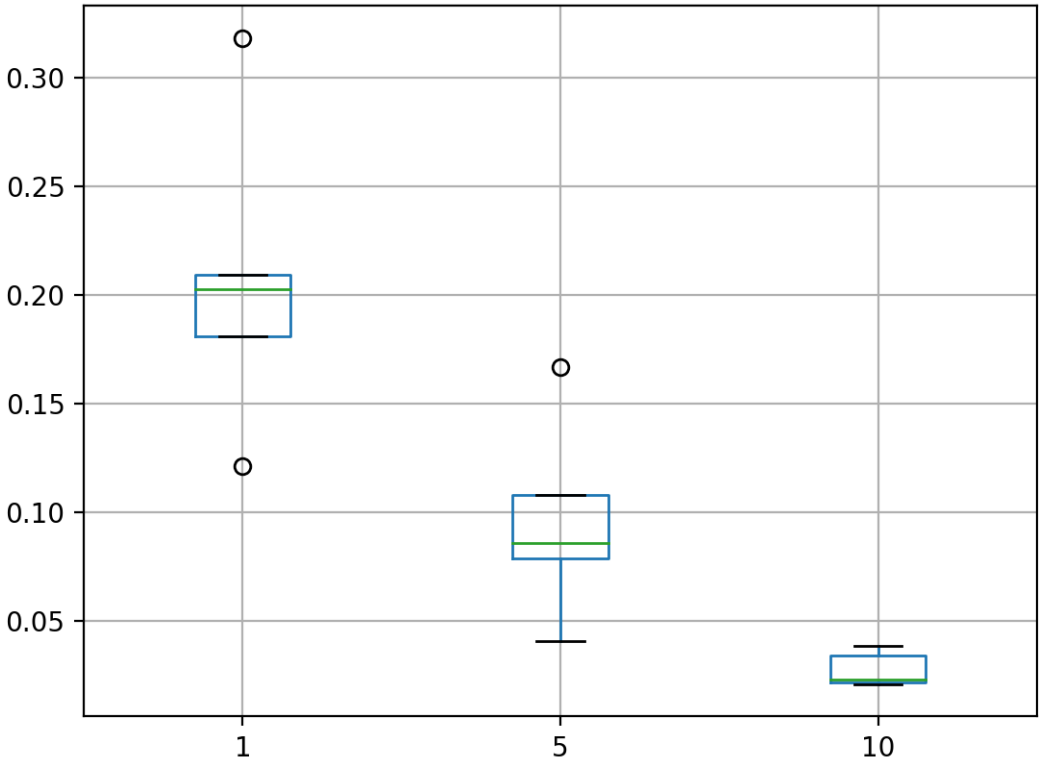

Figure 12.6: Box and whisker plots of the results of tuning the number of memory cells.

12.4.3 Hidden Layers

As with the number of memory cells, we cannot know the *best* number of LSTM hidden layers for a given sequence prediction problem or LSTM architecture. Often deeper is better when you have a lot of data.

- Try grid searching the number of layers and memory cells together.

- Try using patterns of stacking LSTM layers quoted in research papers.
- Try randomly searching the number of layers and memory cells together.

There are patterns of creating deep convolutional neural networks (CNNs). I have not seen any sophisticated patterns for LSTMs beyond the encoder-decoder pattern. I see this as a great open question to explore. Can you devise patterns of stacked LSTMs that work well in general?

12.4.4 Weight Initialization

The Keras `LSTM` layer uses the `glorot_uniform` weight initialization by default. This weight initialization works well in general, but I have had great success using *normal* type weight initialization with LSTMs. Evaluate the effect of different weight initialization schemes on your model skill. Keras offers a great list of weight initialization schemes for you to try. At the very least, compare the skill of these four methods:

- `random_uniform`
- `random_normal`
- `glorot_uniform`
- `glorot_normal`

12.4.5 Activation Functions

The activation function (technically the transfer function as it transfers the weighted activation of the neuron) is often fixed by the framing and scale of the input or output layers. For example, LSTMs use a sigmoid activation function for input and so inputs are often in the scale 0-1. The classification or regression nature of the sequence prediction problem determines the type of activation function to use in the output layer. Challenge the default of sigmoid activation function in the LSTM layers. Try other methods, perhaps in tandem with rescaling input values. For example try:

- `sigmoid`
- `tanh`
- `relu`

Further, challenge whether all LSTM layers in a stacked LSTM need to use the same activation function. In practice, I rarely see a model do better than using sigmoid, but this assumption should be confirmed.

12.5 Tune Learning Behavior

This section outlines some areas of biggest leverage when tuning the learning behavior of your LSTM Model.

12.5.1 Optimization Algorithm

A good default implementation of gradient descent is the Adam algorithm. This is because it automatically uses a custom learning rate for each parameter (weight) in the model, combining the best properties of the AdaGrad and RMSProp methods. Further, the implementation of Adam in Keras uses the best practice initial values for each of the configuration parameters.

Nevertheless, challenge that Adam is the right gradient descent algorithm for your model. Evaluate the performance of models with different gradient descent algorithms. Some ideas of well performing modern algorithms include:

- Adam
- RMSprop
- Adagrad

12.5.2 Learning Rate

The learning rate controls how much to update the weights in response to the estimated gradient at the end of each batch. This can have a large impact on the trade-off between how quickly or how well the model learns the problem. Consider using the classical stochastic gradient descent (SGD) optimizer and explore different learning rate and momentum values. More than just searching values, you can evaluate regimes that vary the learning rate.

- Grid search learning rate values (e.g. 0.1, 0.001, 0.0001).
- Experiment with a learning rate that decays with the number of epochs (e.g. via callback).
- Experiment with updating a fit model with training runs with smaller and smaller learning rates.

The learning rate is tightly coupled with the number of epochs (number of passes through the training samples). Generally, the smaller the learning rate (e.g. 0.0001)), the more training epochs will be required. This is a linear relationship so the reverse is true, where fewer epochs are required for larger learning rates (e.g 0.1).

12.5.3 Batch Size

The batch size is the number of samples between updates to the model weights. A good default batch size is 32 samples.

> [batch size] is typically chosen between 1 and a few hundreds, e.g. [batch size] = 32 is a good default value, with values above 10 taking advantage of the speedup of matrix-matrix products over matrix-vector products.
>
> — *Practical Recommendations For Gradient-based Training Of Deep Architectures*, 2012.

The amount of data and framing of the sequence prediction problem may influence the choice of batch size. Nevertheless, challenge your first best guess and try some alternate configurations

- Batch size of 1 for stochastic gradient descent.
- Batch size of `n`, where `n` is the number of samples for batch gradient descent.
- Grid search batch sizes in powers of 2 from 2 to 256 and beyond.

Larger batch sizes often result in faster convergence of the model, but perhaps to a less optimal final set of weights. A batch size of 1 (stochastic gradient descent) where updates are made after each sample often results in a very noisy learning process. Below is a small example of grid searching the batch sizes 1, 2 and 3 with a small number of repeats (5). You could use this example as a template for your own experiments.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot
from pandas import DataFrame
from numpy import array

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((5, 1, 1))
  return X, y

# return validation data
def get_val():
  seq = [[0.5, 0.6], [0.6, 0.7], [0.7, 0.8], [0.8, 0.9], [0.9, 1.0]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# fit an LSTM model
def fit_model(n_batch):
  # define model
  model = Sequential()
  model.add(LSTM(10, input_shape=(1,1)))
  model.add(Dense(1, activation='linear'))
  # compile model
  model.compile(loss='mse', optimizer='adam')
  # fit model
  X,y = get_train()
  model.fit(X, y, epochs=500, shuffle=False, verbose=0, batch_size=n_batch)
  # evaluate model
  valX, valY = get_val()
  loss = model.evaluate(valX, valY, verbose=0)
  return loss

# define scope of search
params = [1, 2, 3]
n_repeats = 5
# grid search parameter values
```

```
scores = DataFrame()
for value in params:
  # repeat each experiment multiple times
  loss_values = list()
  for i in range(n_repeats):
    loss = fit_model(value)
    loss_values.append(loss)
    print('>%d/%d param=%f, loss=%f' % (i+1, n_repeats, value, loss))
  # store results for this parameter
  scores[str(value)] = loss_values
# summary statistics of results
print(scores.describe())
# box and whisker plot of results
scores.boxplot()
pyplot.show()
```

Listing 12.16: Example of grid searching the batch size.

Running the example prints the progress of the search each iteration. Summary statistics of the results for each configuration are then shown at the end.

```
>1/5 param=1.000000, loss=0.000563
>2/5 param=1.000000, loss=0.001454
>3/5 param=1.000000, loss=0.000777
>4/5 param=1.000000, loss=0.000132
>5/5 param=1.000000, loss=0.001058
>1/5 param=2.000000, loss=0.002671
>2/5 param=2.000000, loss=0.000326
>3/5 param=2.000000, loss=0.002337
>4/5 param=2.000000, loss=0.000697
>5/5 param=2.000000, loss=0.000730
>1/5 param=3.000000, loss=0.001040
>2/5 param=3.000000, loss=0.002549
>3/5 param=3.000000, loss=0.002768
>4/5 param=3.000000, loss=0.017184
>5/5 param=3.000000, loss=0.004616
             1         2         3
count  5.000000  5.000000  5.000000
mean   0.000797  0.001352  0.005631
std    0.000499  0.001070  0.006581
min    0.000132  0.000326  0.001040
25%    0.000563  0.000697  0.002549
50%    0.000777  0.000730  0.002768
75%    0.001058  0.002337  0.004616
max    0.001454  0.002671  0.017184
```

Listing 12.17: Example output of grid searching the batch size.

A box and whisker plot of the final results is created to compare the distribution of model skill for each of the different configurations.

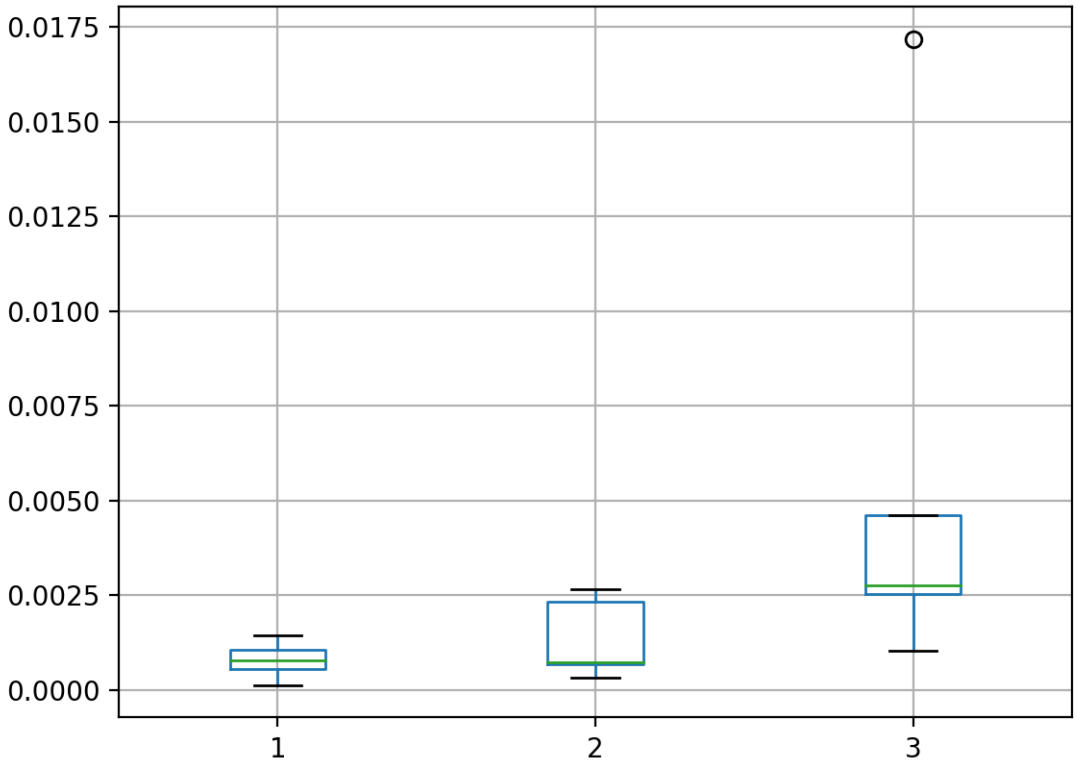

Figure 12.7: Box and whisker plots of the results of tuning the batch size.

12.5.4 Regularization

LSTMs can quickly converge and even overfit on some sequence prediction problems. To counter this, regularization methods can be used. Dropout randomly skips neurons during training, forcing others in the layer to pick up the slack. It is simple and effective. Start with dropout. Dropout rates between 0.0 (no dropout) and 1.0 (complete dropout) can be set on LSTM layers with the two different arguments:

- `dropout`: dropout applied on input connections.
- `recurrent_dropout`: dropout applied to recurrent connections.

For example:

```
model.add(LSTM(..., dropout=0.4))
```

Listing 12.18: Example of adding a layer with input connection dropout.

Some ideas to try include:

- Grid search different dropout percentages.
- Experiment with dropout in the input, hidden, and output layers.

LSTMs also supports other forms of regularization such as weight regularization that imposes pressure to decrease the size of network weights. Again, these can be set on the `LSTM` layer with the arguments:

- `bias_regularizer`: regularization on the bias weights.
- `kernel_regularizer`: regularization on the input weights.
- `recurrent_regularizer`: regularization on the recurrent weights.

Rather than a percentage as in the case of Dropout, you can use a regularization class such as `L1`, `L2`, or `L1L2` regularization. I would recommend using `L1L2` and use values between 0 and 1 that allow to also simulate the `L1` and `L2` approaches. For example, you could try:

- `L1L2(0.0, 0.0)`, e.g. baseline or no regularization.
- `L1L2(0.01, 0.0)`, e.g. `L1`.
- `L1L2(0.0, 0.01)`, e.g. `L2`.
- `L1L2(0.01, 0.01)`, e.g. `L1L2` also called elastic net.

```
model.add(LSTM(..., kernel_regularizer=L1L2(0.01, 0.01)))
```

Listing 12.19: Example of adding a layer with input weight regularization.

In practice, I have found Dropout on the input connections and regularization on input weights separately to result in better performing models.

12.5.5 Early Stopping

The number of training epochs can be very time consuming to tune. An alternative approach is to configure a large number of training epochs. Then setup something to check the performance of the model on train and validation datasets and stop training if it looks like the model is starting to over learn. As such, early stopping is a type of regularization to curb overfitting.

You can experiment with early stopping in Keras with an `EarlyStopping` callback. It requires that you specify a few configuration parameters, such as the metric to monitor (e.g. `val_loss`), the number of epochs over which no improvement in the monitored metric are observed (e.g. 100). A list of callbacks is provided to the `fit()` function when training the model. For example:

```
from keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='val_loss', patience=100)
model.fit(..., callbacks=[es])
```

Listing 12.20: Example of using the `EarlyStopping` callback.

12.6 Further Reading

This section provides some resources for further reading.

12.6.1 Books

- *Empirical Methods for Artificial Intelligence*, 1995.
  http://amzn.to/2tjlD4B

12.6.2 Research Papers

- *Practical recommendations for gradient-based training of deep architectures*, 2012.
  https://arxiv.org/abs/1206.5533
- *Recurrent Neural Network Regularization*, 2014.
  https://arxiv.org/abs/1409.2329

12.6.3 APIs

- `Sequential` Model API in Keras.
  https://keras.io/models/sequential/
- `LSTM` Layer API in Keras.
  https://keras.io/layers/recurrent/#lstm
- Weight Initializers API in Keras.
  https://keras.io/initializers/
- Optimized API in Keras.
  https://keras.io/optimizers/
- Callbacks API in Keras.
  https://keras.io/callbacks/

12.7 Extensions

Do you want to dive deeper into tuning LSTMs? This section lists some challenging extensions to this lesson.

- Select one example from the Models part of the book and develop and summarize robust estimate of the models skill.
- Select one example from the Models part of the book and develop diagnostic plots of the model's performance.
- Select one example from the Models part of the book and tune performance by focusing on the problem framing (e.g. change the prediction model type used, scaling, encoding, etc.). You have to get creative.
- Select one example from the Models part of the book and tune performance by focusing on the model structure (e.g. number of layers, memory cells, etc). You may need to increase the problem difficulty.

- Select one example from the Models part of the book and tune performance by focusing on the model behavior (e.g. number of epochs, training examples, batches, etc.). You may need to increase the problem difficulty.

12.8 Summary

In this lesson, you discovered how to tune LSTM Hyperparameters. Specifically, you learned:

- How to develop a robust evaluation of the skill of your LSTM models.
- How to use learning curves to diagnose the behavior of your LSTM models.
- How to tune the problem framing, structure, and learning behavior of your LSTM models.

In the next lesson, you will discover how to finalize an LSTM model and use it to make predictions on new data.

Chapter 13

How to Make Predictions with LSTMs

13.0.1 Lesson Goal

The goal of this lesson is to learn how to finalize an LSTM model and use it to make predictions on new data. After completing this lesson, you will know:

- How to develop a final LSTM Model for use in your project.
- How to save LSTM Models to file for later use.
- How to make predictions on new data with loaded LSTM Models.

13.0.2 Lesson Overview

This lesson is divided into 3 parts; they are:

1. What Is a Final LSTM Model?
2. Save LSTM Models to File.
3. Make Predictions on New Data.

Let's get started.

13.1 Finalize a LSTM Model

In this section, you will discover how to finalize your LSTM model.

13.1.1 What Is a Final LSTM Model?

A final LSTM model is one that you use to make predictions on new data. That is, given new examples of input data, you want to use the model to predict the expected output. This may be a classification (assign a label) or a regression (a real value). The goal of your sequence prediction project is to arrive at a final model that performs the best, where *best* is defined by:

- **Data**: the historical data that you have available.

- **Time**: the time you have to spend on the project.
- **Procedure**: the data preparation steps, algorithm or algorithms, and the chosen algorithm configurations.

In your project, you gather the data, spend the time you have, and discover the data preparation procedures, algorithm to use, and how to configure it. The final model is the pinnacle of this process, the end you seek in order to start actually making predictions. There is no such thing as a perfect model. There is only the best model that you were able to discover.

13.1.2 What is the Purpose of Using Train/Test Sets?

Creating a train and test split of your dataset is one method to quickly evaluate the performance of an algorithm on your problem. The training dataset is used to prepare a model, to train it. We pretend the test dataset is new data where the output values are withheld from the algorithm. We gather predictions from the trained model on the inputs from the test dataset and compare them to the withheld output values of the test set.

Comparing the predictions to the withheld outputs in the test dataset allows us to compute a performance measure for the model on the test dataset. This is an estimate of the skill of the algorithm trained on the problem when making predictions on unseen data. Using `k`-fold cross-validation is a more robust and more computationally expensive way of calculating this same estimate. We use the estimate of the skill of our LSTM model on a training dataset as a proxy for estimating what the skill of the model will be in practice when making predictions on new data.

This is quite a leap and requires that:

- The procedure that you use is sufficiently robust that the estimate of skill is close to what we actually expect on unseen data.
- The choice of performance measure accurately captures what we are interested in measuring in predictions on unseen data.
- The choice of data preparation is well understood and repeatable on new data, and reversible if predictions need to be returned to their original scale or related to the original input values.
- The choice of model architecture and configuration makes sense for its intended use and operational environment (e.g. complexity).

A lot rides on the estimated skill of the whole procedure on the test set or the `k`-fold cross-validation procedure.

13.1.3 How to Finalize an LSTM Model?

You finalize a model by applying the chosen LSTM architecture and configuration on all of your data. There is no train and test split and no cross-validation folds. Put all of the data back together into one large training dataset and fit your model. That's it. With the finalized model, you can:

- Save the model for later or operational use.
- Load the model and make predictions on new data.

Why Not Keep the Best Trained Model?

It is possible that your LSTM model takes many days or weeks to prepare. In that case, you may want to keep the model fit on the train dataset without fitting it on the combination of train and test sets. This is a trade-off between the possible benefits of training the model on the additional data and the time and computational cost for fitting a new model.

Won't the Performance of the Final Model Be Different?

The whole idea of using a robust test harness was to estimate the skill of the final model. Ideally, the difference in skill between the estimate and what is observed in the final model is minor to the point of measurement error or that the skill is lifted as a function of the number of training examples used to fit the model. You can test both of these assumptions by performing a sensitivity analysis of model skill versus number of training examples.

Won't the Final Model Be Different Each Time it Is Trained?

The estimate of skill that you used to choose the final model should be averaged over multiple runs. That way, you know that on average the chosen model architecture and configuration is skillful. You could try to control for the stochastic nature of the model by training multiple final models and using an ensemble or average of their predictions in practice. Again, you can design a sensitivity analysis to test whether this will result in a more stable set of predictions.

13.2 Save LSTM Models to File

Keras provides an API to allow you to save your model to file. There are two options:

1. Save model to a single file.
2. Save architecture and weights to separate files.

In both cases, the HDF5 file format is used that efficiently stores large arrays of numbers on disk. You will need to confirm that you have the `h5py` Python library installed. It can be installed as follows:

```
sudo pip install h5py
```

Listing 13.1: Install the required `h5py` Python library.

13.2.1 Save to Single File

You can save a fit Keras model to file using the `save()` function on the model. For example:

```
# define model
model = Sequential()
model.add(LSTM(...))
# compile model
model.compile(...)
# fit model
model.fit(...)
# save model to single file
model.save('lstm_model.h5')
```

Listing 13.2: Example of saving a fit LSTM model to a single file.

This single file will contain the model architecture and weights. It also includes the specification of the chosen loss and optimization algorithm so that you can resume training. The model can be loaded again (from a different script in a different Python session) using the `load_model()` function.

```
from keras.models import load_model
# load model from single file
model = load_model('lstm_model.h5')
# make predictions
yhat = model.predict(X, verbose=0)
print(yhat)
```

Listing 13.3: Example of loading a saved LSTM model from a single file.

Below is a complete example of fitting an LSTM model, saving it to a single file and later loading it again. Although the loading of the model is in the same script, this section may be run from another script in another Python session. Running the example saves the model to the file `lstm_model.h5`.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from numpy import array
from keras.models import load_model

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# define model
model = Sequential()
model.add(LSTM(10, input_shape=(1,1)))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mse', optimizer='adam')
# fit model
X,y = get_train()
model.fit(X, y, epochs=300, shuffle=False, verbose=0)
# save model to single file
model.save('lstm_model.h5')
```

```
# snip...
# later, perhaps run from another script

# load model from single file
model = load_model('lstm_model.h5')
# make predictions
yhat = model.predict(X, verbose=0)
print(yhat)
```

Listing 13.4: Example of saving a fit LSTM model to a single file and later loading it again.

13.2.2 Save to Separate Files

You can save the model architecture (e.g. layers and how they connect) and weights (arrays of numbers) to separate files. I recommend this approach as it allows you to develop updated model weights and replace one file while ensuring the model architecture is left unchanged.

Save Architecture

Keras provides two formats for preserving the model architecture: JSON and YAML formats. The benefit of these formats is that they are human readable. The choice is really a matter of taste or preference. The `to_json()` or `to_yaml()` functions on the fit model can be called to save your model to the JSON or YAML formats respectively. They return a formatted string that you can then save to disk using the standard Python files `open()` and `write()` functions for saving an ASCII file.

```
...
# convert model architecture to JSON format
architecture = model.to_json()
# save architecture to JSON file
with open('architecture.json', 'wt') as json_file:
    json_file.write(architecture)
```

Listing 13.5: Example of saving a fit LSTM model architecture to a file.

The model architecture can be loaded again (from a different script in a different Python session) using the `model_from_json()` or `model_from_yaml()` functions. Once loaded, the `model_from_json()` or `model_from_yaml()` functions can be used to create a Keras model from the architecture.

```
from keras.models import model_from_json

# load architecture from JSON File
json_file = open('architecture.json', 'rt')
architecture = json_file.read()
json_file.close()
# create model from architecture
model = model_from_json(architecture)
```

Listing 13.6: Example of loading a saved LSTM model architecture from a file.

Save Weights

Keras provides a function on a fit model to save model weights. The `save_weights()` function will save model weights to an HDF5 formatted file.

```
...
# save weights to hdf5 file
model.save_weights('weights.h5')
```

Listing 13.7: Example of saving a fit LSTM model weights to a file.

The model weights can be loaded again (from a different script in a different Python session) using the `load_weights()` function on the model object. This means that you must already have a model, either created anew or created from a loaded architecture.

```
model = ...
# load weights from hdf5 file
model.load_weights('weights.h5')
```

Listing 13.8: Example of loading a saved LSTM model weights from a file.

Example

We can put this together into a single worked example. The demonstration below fits an LSTM model on a small contrived dataset. The model architecture is saved in JSON format and the model weights are stored in HDF5 format. You can easily change the example to save the architecture in YAML format if you prefer. The model is then loaded from these files and used to make a prediction.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from numpy import array
from keras.models import model_from_json

# return training data
def get_train():
  seq = [[0.0, 0.1], [0.1, 0.2], [0.2, 0.3], [0.3, 0.4], [0.4, 0.5]]
  seq = array(seq)
  X, y = seq[:, 0], seq[:, 1]
  X = X.reshape((len(X), 1, 1))
  return X, y

# define model
model = Sequential()
model.add(LSTM(10, input_shape=(1,1)))
model.add(Dense(1, activation='linear'))
# compile model
model.compile(loss='mse', optimizer='adam')
# fit model
X,y = get_train()
model.fit(X, y, epochs=300, shuffle=False, verbose=0)
# convert model architecture to JSON format
architecture = model.to_json()
# save architecture to JSON file
```

```
with open('architecture.json', 'wt') as json_file:
    json_file.write(architecture)
# save weights to hdf5 file
model.save_weights('weights.h5')

# snip...
# later, perhaps run from another script

# load architecture from JSON File
json_file = open('architecture.json', 'rt')
architecture = json_file.read()
json_file.close()
# create model from architecture
model = model_from_json(architecture)
# load weights from hdf5 file
model.load_weights('weights.h5')
# make predictions
yhat = model.predict(X, verbose=0)
print(yhat)
```

Listing 13.9: Example of saving a fit LSTM model to separate files and later loading it again.

Running this example saves the model architecture to the file `architecture.json` and the weights to the file `weights.h5`. The model is then loaded from these same files. Although the model loading is demonstrated in the same script, you could just as easily run that section from another script in a different Python session.

13.3 Make Predictions on New Data

After you have finalized your model and saved it to file, you can load it and use it to make predictions.

- On a sequence regression problem, this may be the prediction of the real value at the next time step.
- On a sequence classification problem, this may be a class outcome for a given input sequence.

Or it may be any other variation based on the specifics of your sequence prediction problem. You would like an outcome from your model (`yhat`) given an input sequence (`X`) where the true outcome for the sequence (`y`) is currently unknown. You may be interested in making predictions in a production environment, as the backend to an interface, or manually. It really depends on the goals of your project. Making predictions on new data involves two key steps:

1. Data Preparation.
2. Predicting Values.

13.3.1 Data Preparation

Any data preparation performed on your training data prior to fitting your final model must also be applied to any new data prior to making predictions. For example, if your training data was normalized, then all new data for which you would like a prediction must also be normalized. In turn, this means that the coefficients used to normalize the data (e.g. min and max values) too must be saved as part of finalizing your model and later loaded when a prediction is needed.

This applies to other forms of data preparation such as log transforms, standardizing values, and making a series stationary. The specific way that raw data is transformed into its vectorized form must also be duplicated. This includes the padding or truncating of sequences and, importantly, the reshaping of raw sequences into the 3D format of samples, time steps, and features.

Depending on the framing of your problem, the number of samples when predicting may refer to the number of input sequences for which you would like an output prediction. It may be 1 for a single prediction. Think of all the data preparation performed in transforming raw data to the data used to train the final model as a pipeline. This pipeline must also be applied any time a prediction is to be made.

13.3.2 Predicting Values

Predicting is the easy part. We covered the basic API in Chapter 4, but we will review it again in more detail. It involves taking the prepared input data (`X`) and calling one of the Keras prediction methods on the loaded model. Remember that the input for making a prediction (`X`) is only comprised of the input sequence data required to make a prediction, not all prior training data. In the case of predicting the next value in one sequence, the input sequence would be 1 sample with the fixed number of time steps and features used when you defined and fit your model.

For example, a raw prediction in the shape and scale of the activation function of the output layer can be made by calling the `predict()` function on the model:

```
X = ...
model = ...
yhat = model.predict(X)
```

Listing 13.10: Example of making a prediction with a fit LSTM model on new data.

The prediction of a class index can be made by calling the `predict_classes()` function on the model.

```
X = ...
model = ...
yhat = model.predict_classes(X)
```

Listing 13.11: Example of predicting classes with a fit LSTM model on new data.

The prediction of probabilities can be made by calling the `predict_proba()` function on the model.

```
X = ...
model = ...
yhat = model.predict_proba(X)
```

Listing 13.12: Example of predicting probabilities with a fit LSTM model on new data.

13.4 Further Reading

13.4.1 API

- How can I save a Keras model? in the Keras FAQ.
  https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model
- Save and Load Keras API.
  https://keras.io/models/about-keras-models/

13.5 Extensions

Do you want to dive deeper into finalizing an LSTM model? This section lists some challenging extensions to this lesson.

- List one or more prediction problems where you would like to develop an LSTM model and save it for later use in making predictions.
- Update an example from the Models part of the book to save the model to a multiple files, then load it again from another script and make predictions.
- Update an example from the Models part of the book with a classification output to predict probabilities, and present the probabilities, perhaps graphically.
- Update an example from the Models part of the book with a classification output to use the `predict()` function, then use the `argmax()` function to interpret the results as class values.
- Select an example from the Models part of the book, update it save the model and data preparation information to file (e.g. scaling or encoding information). Load the model and data preparation information and use the model to make predictions on new data that must be prepared prior to making a prediction.

13.6 Summary

In this lesson, you discovered how to make best use of new data in updating your finalized LSTM models. Specifically, you learned:

- How to develop a final LSTM Model for use in your project.
- How to save LSTM Models to file for later use.
- How to make predictions on new data with loaded LSTM Models.

In the next lesson, you will discover how to update finalized LSTM models in order to make the best use of new data.

Chapter 14

How to Update LSTM Models

14.0.1 Lesson Goal

The goal of this lesson is to learn how to update LSTM models after new data becomes available. After completing this lesson, you will know:

- The interest in monitoring, recovering skill, and lifting model skill with new data.
- The 5-step process to updating a finalized LSTM model with new data.
- The 4 key approaches to consider when developing an updated LSTM model with new data.

14.0.2 Lesson Overview

This lesson is divided into 3 parts; they are:

1. What About New Data?
2. What Is LSTM Model Updating?
3. 5-Step Process to Update LSTM Models.

Let's get started.

14.1 What About New Data?

Once you finalize your LSTM model, you can use it to make predictions. But that is not the end of the story. After months or years, you will start to accumulate a corpus of new data. This will raise some important questions. The first of which is:

14.1.1 Is the model still skillful?

It is important to have a handle on this question as part of the ongoing maintenance of your model. Once you're monitoring model skill, perhaps you notice that the skill of predictions is decreasing over time.

14.1.2 Can we recover model skill?

The nature of the sequence prediction problem addressed by your LSTM model may change over time. A model is only as good as the data used to train it. If the data used to train your model was from one year ago, perhaps that new data collected to day would result in a different and more skillful model. Perhaps your model predictions are just as skillful as when you first developed the model.

14.1.3 Can we lift the model skill?

It may be possible to lift the skill of your model by making use of this new data.

14.2 What Is LSTM Model Updating?

Updating LSTM models refers to techniques used to make best use of new data to evaluate and improve the skill of an existing and already finalized LSTM model. The goal is to evaluate whether new or updated candidate models are more skillful than the existing finalized model. This does not mean tuning the existing model on the old training dataset. It explicitly refers to how to best incorporate new data into the updating of the existing model which may or may not involve tuning model hyperparameters. Like all modeling, it is important to be systematic when updating LSTM models.

14.3 5-Step Process to Update LSTM Models

Updating an existing LSTM model involves 5 key steps. They are:

1. Collect New Data.
2. Evaluate Existing Model.
3. Develop Updated Models.
4. Evaluate Updated Models.
5. Replace Model.

Let's take a closer look at each in turn.

14.3.1 Collect New Data

Without new data, you cannot update your model. This means complete and high-quality input sequences in the case of sequence regression or complete input sequences and their associated class labels in the case of sequence classification.

If the sequence prediction problem has not changed, the data should have the same format and be prepared in the same way as was done in the development of the original model. Ideally, it would also be valuable to have access to the data used to train the existing model.

You may have months or years of data, perhaps more than you can handle. If this is the case, consider reusing the methods you used to select data to train and evaluate the existing

model. Consider selecting a subsample of sequences, perhaps the most recent, perhaps a portion from each time interval. Having access to a lot of data (even too much) can be useful. We may or may not decide to use some in the updating of the model (a slow process). But we could use most or all of it to evaluate the existing and new candidate models (a fast process).

14.3.2 Evaluate Existing Model

It is critical to monitor the skill of the predictions of your model. As with evaluating candidate models, evaluating the performance of a finalized model requires having access to both the predictions and the true observations. Specifically, this means that:

1. **Predictions**: You must store some or all predictions made by the finalized model.
2. **Observations**: You must gather and store some or all of the true or actual observations.

Given model predictions and observations, you can evaluate the skill of the finalized model on recent data and quantify it by an appropriate interval, such as hour, day, week, or month. It is useful to have a picture of model skill over a long period of time, even back to the inception of the model. This may require back-testing the model and regenerating any predictions that were not stored.

Plotting model skill over time will help you answer the question as to whether model skill is holding steady or degrading. This in turn informs you as to whether your project is that of maintenance to recover model skill or development to improve model skill.

- **Degrading Model Skill**: Your goal is to recover model skill back to better than historical levels by making use of new data.
- **Stable Model Skill**: Your goal is to lift model skill over the stable level by making use of new data.
- **Improving Model Skill**: Great! Your goal may be to investigate what has changed in the new data compared to the old data used to fit the original model.

14.3.3 Develop Updated Models

There are many ways to make use of new data. Below are 4 options for you to consider.

- **Update model on new data**. The existing model is loaded and trained on additional epochs using only the new data.
- **Update model on old and new data**. The existing model is loaded and trained on additional epochs using a mixture of the original data used to fit the model (old data and new data).
- **Develop a new model on new data**. A new model is developed from scratch and fit only on the new data.
- **Develop a new model on new and old data**. A new model is developed and fit on the old and new data.

The specific approach you choose may depend on your sequence prediction problem, your specific implementation, or, ultimately, the skill of the candidate models. Updating a model is as simple as loading the model and running additional training epochs. Internally, the model is defined by a structure (how things hang together) and weights (arrays of numbers). Generally, updating considers the work of finding a good network structure to have been addressed and that we are primarily interested in a refinement process of the network weights. For example, below is a snippet for how weight updating may look in Keras:

```
# load model from file
model = ...
# access new data
newX, newY = ...
# fit model on new data
model.fit(newX, newY, ...)
```

Listing 14.1: Example of updating an existing LSTM model.

Tuning Updated Models

There are tuning options when updating a model that you may want to consider. For example:

- **Learning rate**: perhaps a small learning rate is required to make small adjustments to the weights rather than large jumps.
- **Epochs**: perhaps a small number of iterations over the new sequences are required to dial in the weights to the new data.
- **Samples**: perhaps only the most recent samples from the last day, week, month, or year are required to dial in the model.

Consider grid searching over a list of these and other concerns with the focus on tuning the weights to the new data, rather than wholesale replacing them with new and vastly different weights.

Tuning New Models

If you decide to explore developing a new model from scratch, the whole suite of model selection and hyperparameter tuning is open to you. This can be daunting, especially for the first *update project*. Consider leaving the model structure fixed and focus on developing a new set of weights by making use of the new data.

Perhaps focus efforts on which and how many data samples to use to fit the model. I would recommend exploring a sensitivity analysis of model skill versus the scope of recent data used for training (e.g. a model fit on last month's data, one fit on the last 3 months, etc.).

14.3.4 Evaluating Candidate Models

The updated models are in fact new candidate models to replace the existing model. As such, these models must be rigorously evaluated and compared to the existing model.

Evaluation Data

This means making predictions on the same data used to evaluate the existing model and interpreting the skill scores to see how the candidate models perform. The skill does not have to be compared for all time; in fact, it probably should not. I would recommend focusing on evaluating and comparing model skill on sequences from a recent interval, such as the last month, 3 months, or 6 months, depending on the problem and data availability.

Robust Evaluation

The skill scores are relative. The existing model is the baseline and improving over the baseline is the goal. This means that the results must be robust.

- Repeat the experiments over a large corpus of test data to control for the variance in the stochastic data.
- Repeat experiments multiple times to control for the variance in the stochastic algorithms.
- Consider the use of statistical tests to inform you as to whether the difference between two populations of results is significant and to what degree.

Deciding whether or not to replace an existing model with a new model must be a strongly defended decision and the only defense you have is robust results.

Fair Baseline

The candidates must robustly outperform the existing model. The existing model is the baseline. But, it is important to allow the existing model to provide a fair point of comparison. If you have developed models that update the existing weights with `n` additional epochs on new data, then consider including a candidate model that updates the existing model on old data for `n` additional epochs. This and similar points of comparison will help you tease out whether any change in model skill is due to the additional epochs or can be credited to the new data.

Present Results

The decision may not be yours alone, e.g. other stakeholders. Consider presenting results using charts like box-and-whisker plots that allow you to visually compare the distribution of results, including the mean, median, and other percentiles. Also consider presenting the results not in terms of model skill (e.g. loss or accuracy), but the effect the change in skill has on users, experience, costs, or other business concerns.

14.3.5 Replace Model

Once you have evaluated the candidate models, you may have a new model to replace the existing model. Given sign-off from stakeholders, this should be a straightforward process. I recommend storing model weights and model structure in separate configuration files. This allows you to update model weights alone when updating models, which is a smaller and less risky change.

It is important to monitor the skill of the new model and the old model in parallel for some time going forward. This ongoing monitoring of the current and previous models is critical.

- It allows you to defend the decision of updating the model if all is well.
- It gives you evidence as to whether the previous model should be switched back if all is not well.
- It helps in reporting to stakeholders on the health and ongoing improvement of the system.

14.4 Extensions

Do you want to dive deeper into updating a fit LSTM model? This section lists some challenging extensions to this lesson.

- List 5 sequence prediction problems that if were already modeled could be improved with new data.
- List 5 considerations that a team and stakeholders may want to review before replacing an existing model in production with a new model.
- Research the topic of *concept drift* and describe the implications for one sequence prediction problem.
- Outline a strategy that you could use to continually monitor the skill of a deployed LSTM model given a stream of new data for which it must make hourly predictions and for which truth values are made available the next day.
- Select one example from the Models part of the book and go through the model updating procedure process with new data that varies systematically from the data used to fit the original model.

14.5 Summary

In this lesson, you discovered how to make best use of new data in updating your finalized LSTM models. Specifically, you learned:

- The interest in monitoring, recovering skill, and lifting model skill with new data.
- The 5-step process to updating a finalized LSTM model with new data.
- The 4 key approaches to consider when developing an updated LSTM model with new data.

This was the final lesson, well done!

Part V

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with LSTMs for sequence prediction with Keras in Python. As you start to work on projects and expand your existing knowledge of the techniques you may need help. This appendix points out some of the best sources of help.

A.1 Official Keras Destinations

This section lists the official Keras sites that you may find helpful.

- Keras Official Blog.
  https://blog.keras.io/
- Keras API Documentation.
  https://keras.io/
- Keras Source Code Project.
  https://github.com/fchollet/keras

A.2 Where to Get Help with Keras

This section lists the 9 best places I know where you can get help with Keras and LSTMs.

- Keras Users Google Group.
  https://groups.google.com/forum/#!forum/keras-users
- Keras Slack Channel (you must request to join).
  https://keras-slack-autojoin.herokuapp.com/
- Keras on Gitter.
  https://gitter.im/Keras-io/Lobby#
- Keras tag on StackOverflow.
  https://stackoverflow.com/questions/tagged/keras
- Keras tag on CrossValidated.
  https://stats.stackexchange.com/questions/tagged/keras

- Keras tag on DataScience.
  https://datascience.stackexchange.com/questions/tagged/keras
- Keras Topic on Quora.
  https://www.quora.com/topic/Keras
- Keras Github Issues.
  https://github.com/fchollet/keras/issues
- Keras on Twitter.
  https://twitter.com/hashtag/keras

A.3 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like *my model does not work* or *how does x work*.
- Search for answers before asking questions.
- Provide complete code and error messages.
- Boil your code down to the smallest possible working example that demonstrates the issue.

A.4 Contact the Author

You are not alone. If you ever have any questions about LSTMs or this book, please contact me directly. I will do my best to help.

Jason Brownlee
Jason@MachineLearningMastery.com

Appendix B

How to Setup a Workstation for Deep Learning

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning and deep learning software. These instructions are suitable for Windows, Mac OS X, and Linux platforms. I will demonstrate them on OS X, so you may see some mac dialogs and file extensions.

B.1 Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda
4. Install Deep Learning Libraries

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
  https://www.continuum.io/
- 2. Click `Anaconda` from the menu and click `Download` to go to the download page.
  https://www.continuum.io/downloads

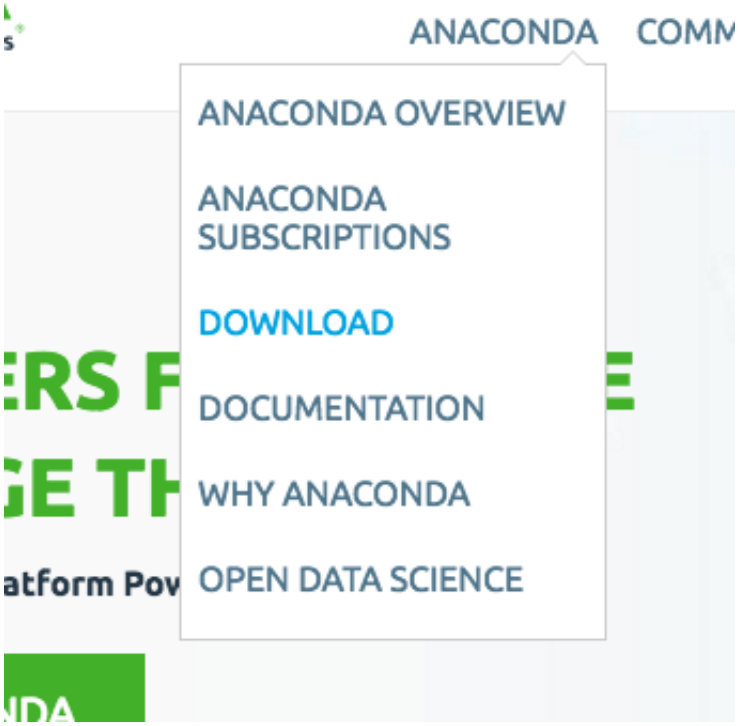

Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
  - Choose Python 3.6
  - Choose the Graphical Installer

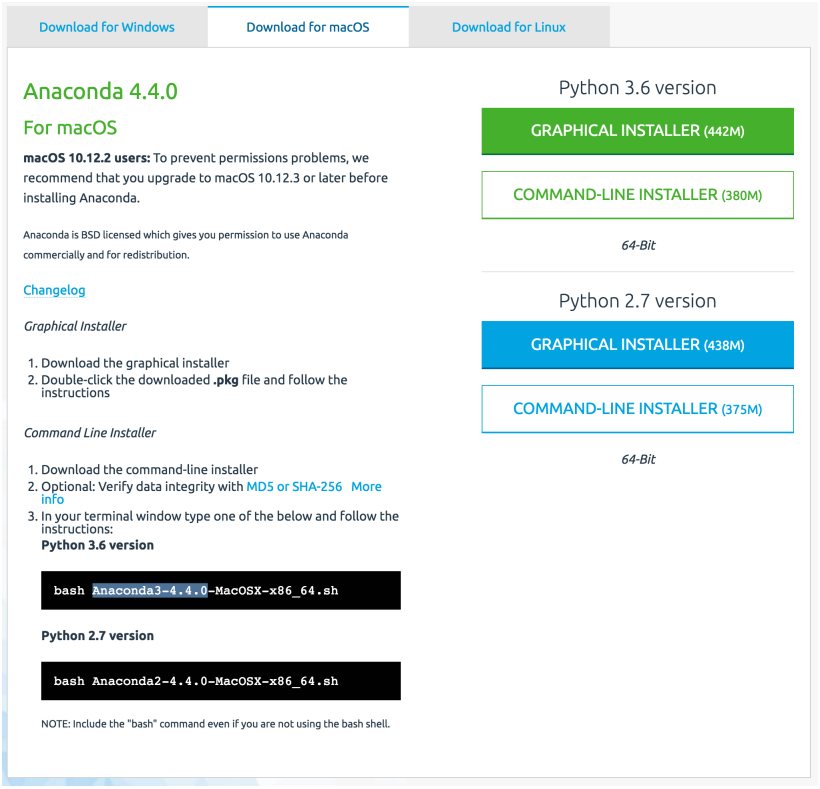

Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on OS X, so I chose the OS X version. The file is about 426 MB. You should have a file with a name like:

```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on Mac OS X.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.

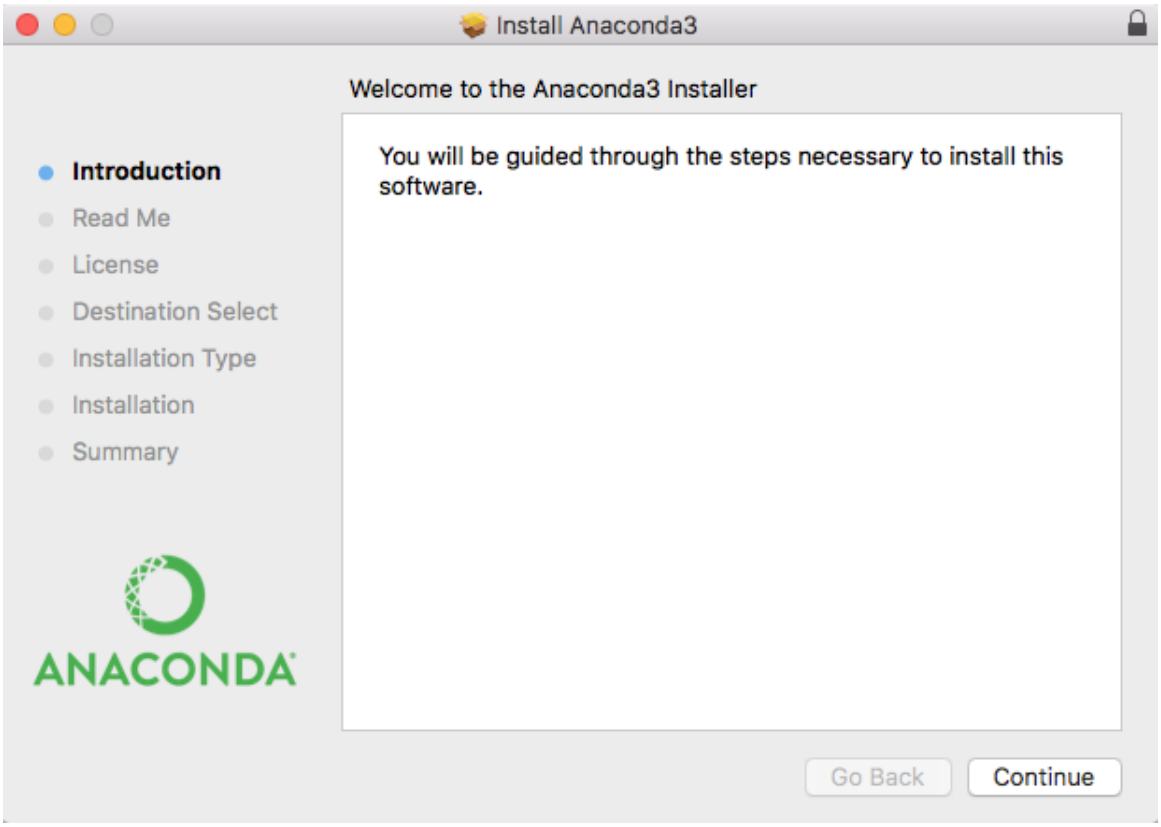

Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.

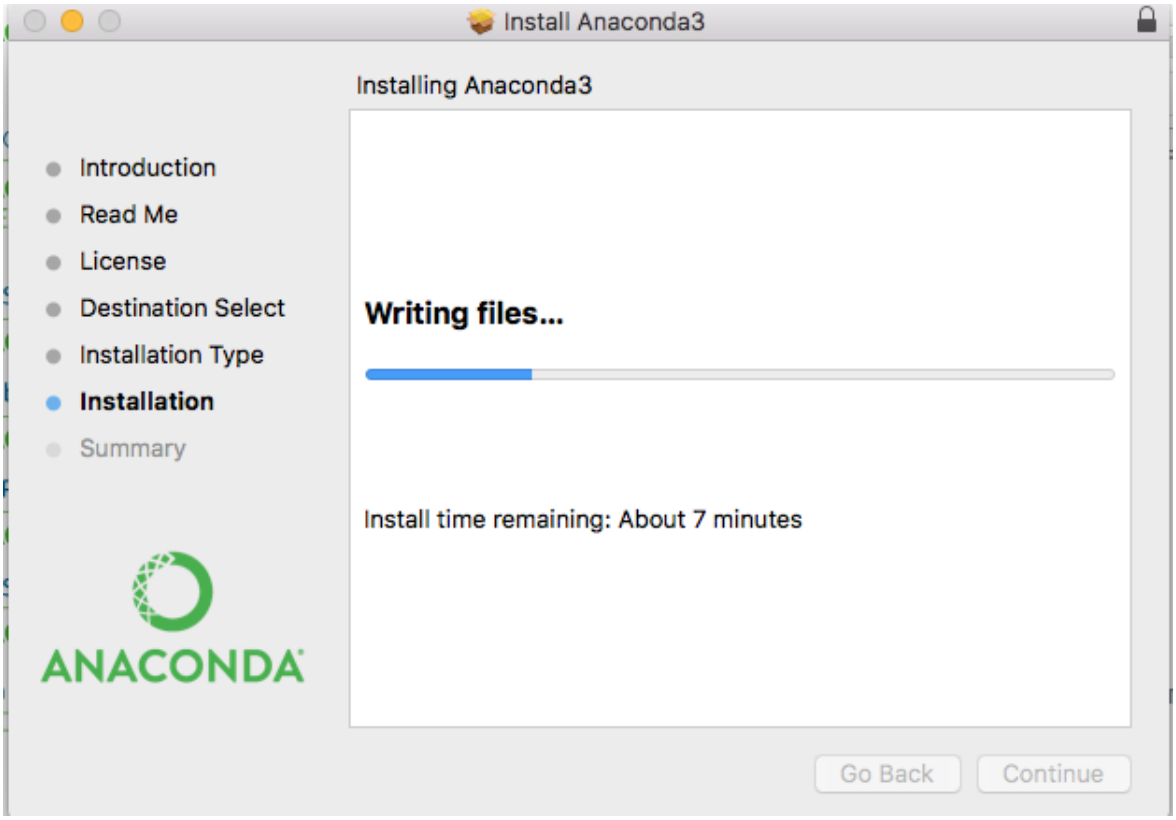

Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

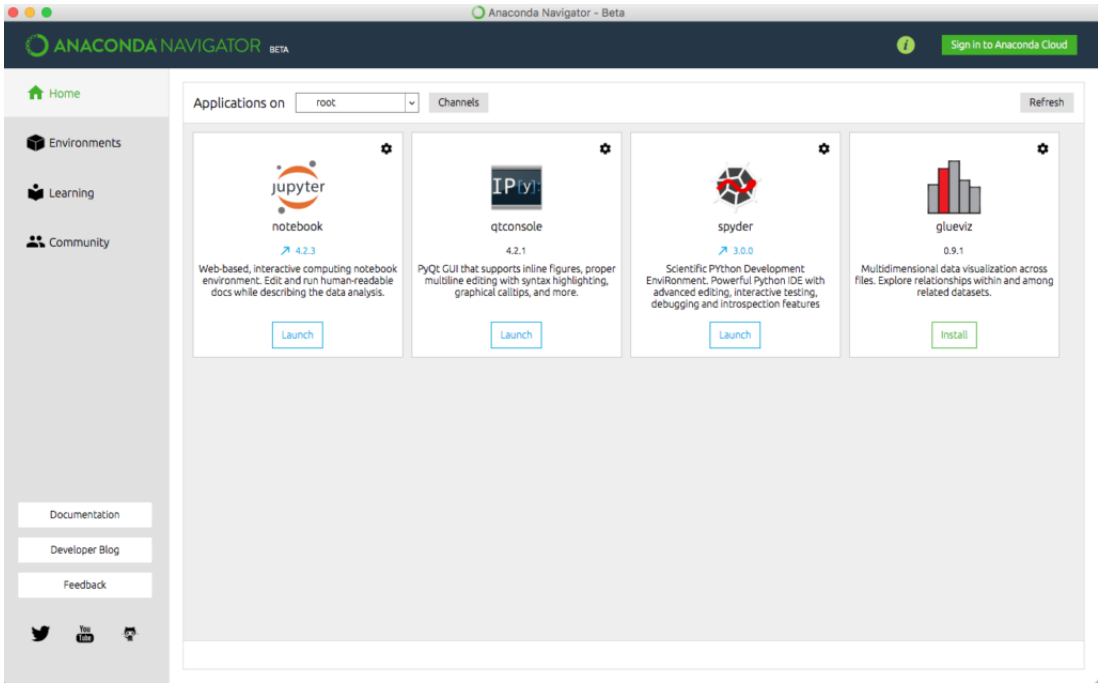

Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -V
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -V
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.3.1
numpy: 1.17.2
matplotlib: 3.1.1
pandas: 0.25.1
statsmodels: 0.10.1
sklearn: 0.21.3
```

Listing B.9: Sample output of versions script.

B.5 Install Deep Learning Libraries

In this step, we will install Python libraries used for deep learning, specifically: Theano, TensorFlow, and Keras. NOTE: I recommend using Keras for deep learning and Keras only requires one of Theano or TensorFlow to be installed. You do not need both. There may be problems installing TensorFlow on some Windows machines.

- 1. Install the Theano deep learning library by typing:

```
conda install theano
```

Listing B.10: Install Theano with conda.

- 2. Install the TensorFlow deep learning library by typing:

```
conda install -c conda-forge tensorflow
```

Listing B.11: Install TensorFlow with conda.

Alternatively, you may choose to install using `pip` and a specific version of TensorFlow for your platform.

- 3. Install Keras by typing:

```
pip install keras
```

Listing B.12: Install Keras with `pip`.

- 4. Confirm your deep learning environment is installed and working correctly.

Create a script that prints the version numbers of each library, as we did before for the SciPy environment.

```
# theano
import theano
print('theano: %s' % theano.__version__)
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
# keras
import keras
print('keras: %s' % keras.__version__)
```

Listing B.13: Code to check that key deep learning libraries are installed.

Save the script to a file `deep_versions.py`. Run the script by typing:

```
python deep_versions.py
```

Listing B.14: Run script from the command line.

You should see output like:

```
theano: 1.0.4
tensorflow: 2.0.0
keras: 2.3.0
```

Listing B.15: Sample output of the deep learning versions script.

B.6 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
  https://www.continuum.io/
- Anaconda Navigator.
  https://docs.continuum.io/anaconda/navigator.html
- The conda command line tool.
  http://conda.pydata.org/docs/index.html
- Instructions for installing TensorFlow in Anaconda.
  https://www.tensorflow.org/get_started/os_setup#anaconda_installation

B.7 Summary

Congratulations, you now have a working Python development environment for machine learning and deep learning. You can now learn and practice machine learning and deep learning on your workstation.

Appendix C

How to Use Deep Learning in the Cloud

Large deep learning models require a lot of compute time to run. You can run them on your CPU but it can take hours or days to get a result. If you have access to a GPU on your desktop, you can drastically speed up the training time of your deep learning models. In this project you will discover how you can get access to GPUs to speed up the training of your deep learning models by using the Amazon Web Service (AWS) infrastructure. For less than a dollar per hour and often a lot cheaper you can use this service from your workstation or laptop. After working through this project you will know:

- How to create an account and log-in to Amazon Web Service.
- How to launch a server instance for deep learning.
- How to configure a server instance for faster deep learning on the GPU.

Let's get started.

C.1 Project Overview

The process is quite simple because most of the work has already been done for us. Below is an overview of the process.

- Setup Your AWS Account.
- Launch Your Server Instance.
- Login and Run Your Code.
- Close Your Server Instance.

Note, it costs money to use a virtual server instance on Amazon. The cost is low for model development (e.g. less than one US dollar per hour), which is why this is so attractive, but it is not free. The server instance runs Linux. It is desirable although not required that you know how to navigate Linux or a Unix-like environment. We're just running our Python scripts, so no advanced skills are needed.

Note: The specific versions may differ as the software and libraries are updated frequently.

C.2 Setup Your AWS Account

You need an account on Amazon Web Services[1].

- 1. You can create account by the Amazon Web Services portal and click *Sign in to the Console*. From there you can sign in using an existing Amazon account or create a new account.

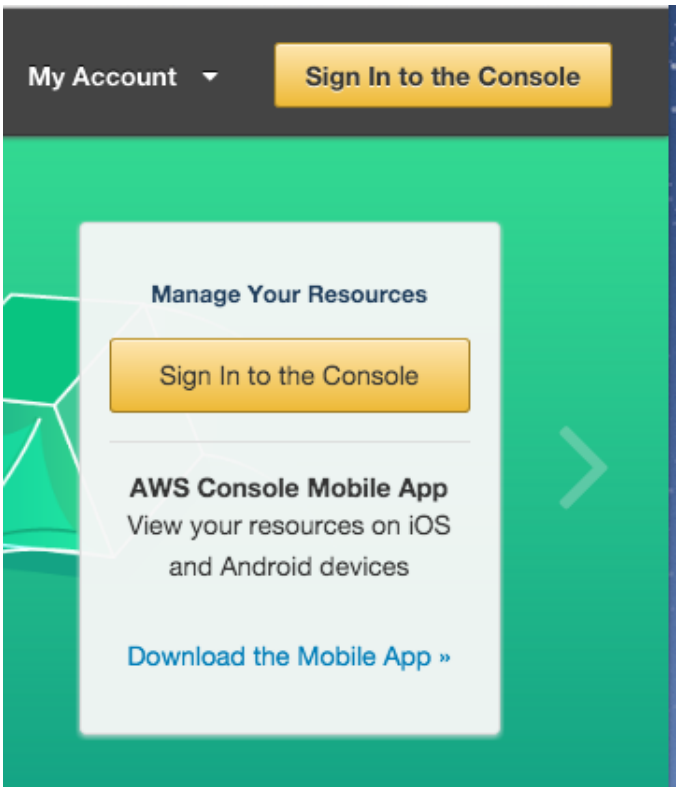

Figure C.1: AWS Sign-in Button

- 2. You will need to provide your details as well as a valid credit card that Amazon can charge. The process is a lot quicker if you are already an Amazon customer and have your credit card on file.

[1] https://aws.amazon.com

Figure C.2: AWS Sign-In Form

Once you have an account you can log into the Amazon Web Services console. You will see a range of different services that you can access.

C.3 Launch Your Server Instance

Now that you have an AWS account, you want to launch an EC2 virtual server instance on which you can run Keras. Launching an instance is as easy as selecting the image to load and starting the virtual server. Thankfully there is already an image available that has almost everything we need it is called the **Deep Learning AMI Amazon Linux Version** and was created and is maintained by Amazon. Let's launch it as an instance.

- 1. Login to your AWS console if you have not already.
  https://console.aws.amazon.com/console/home

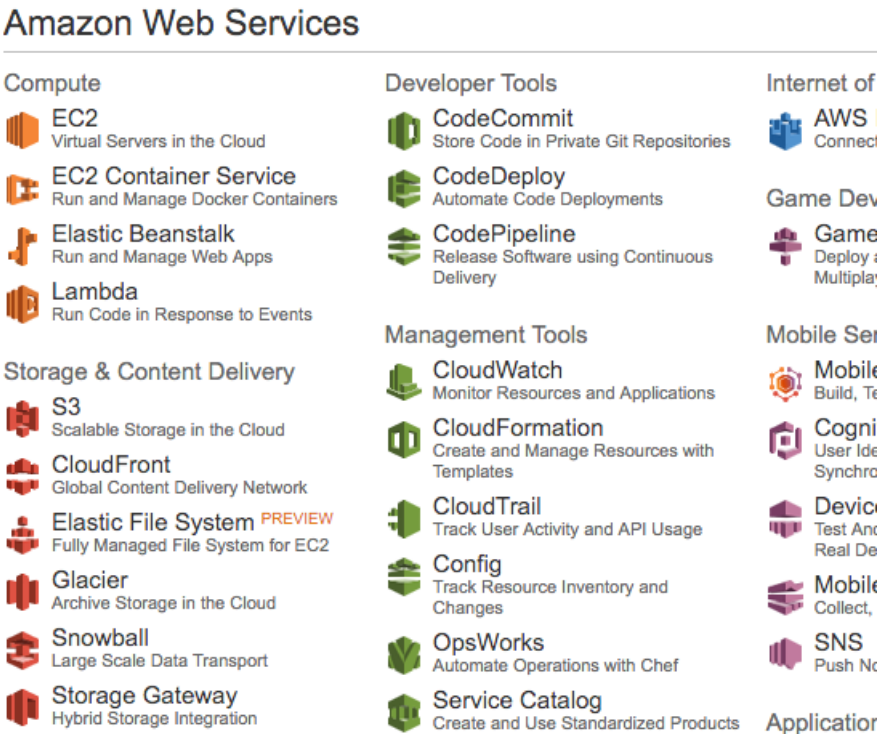

Figure C.3: AWS Console

- 2. Click on EC2 for launching a new virtual server.
- 3. Select *US West Orgeon* from the drop-down in the top right hand corner. This is important otherwise you will not be able to find the image we plan to use.
- 4. Click the *Launch Instance* button.
- 5. Click *Community AMIs*. An AMI is an Amazon Machine Image. It is a frozen instance of a server that you can select and instantiate on a new virtual server.

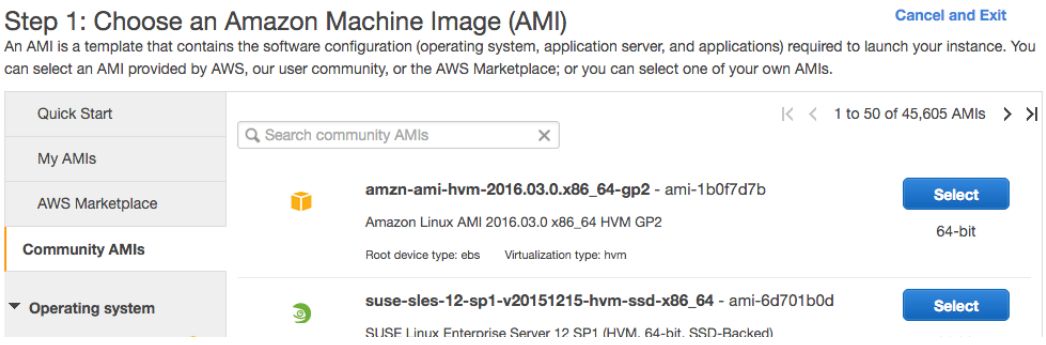

Figure C.4: Community AMIs

- 6. Enter `ami-dfb13ebf` in the *Search community AMIs* search box and press enter (this is the current AMI id for v2.0 but the AMI may have been updated since, you check for a more recent id[2]). You should be presented with a single result.

[2] https://aws.amazon.com/marketplace/pp/B01M0AXXQB

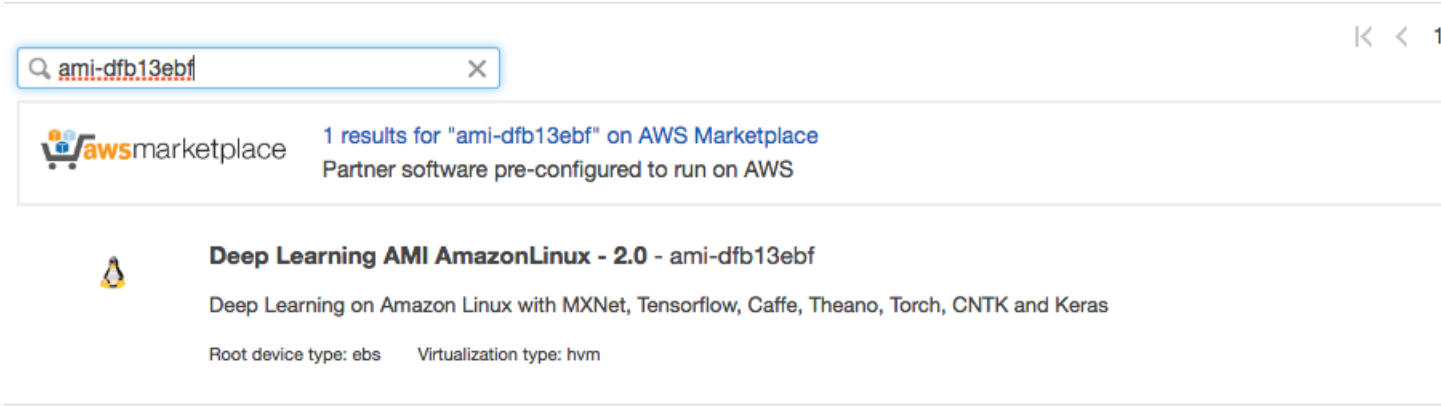

Figure C.5: Select a Specific AMI

- 7. Click *Select* to choose the AMI in the search result.
- 8. Now you need to select the hardware on which to run the image. Scroll down and select the *g2.2xlarge* hardware. This includes a GPU that we can use to significantly increase the training speed of our models.

Figure C.6: Select g2.2xlarge Hardware

- 9. Click *Review and Launch* to finalize the configuration of your server instance.
- 10. Click the *Launch* button.
- 11. Select Your Key Pair.

If you have a key pair because you have used EC2 before, select *Choose an existing key pair* and choose your key pair from the list. Then check *I acknowledge....* If you do not have a key pair, select the option *Create a new key pair* and enter a *Key pair name* such as keras-keypair. Click the *Download Key Pair* button.

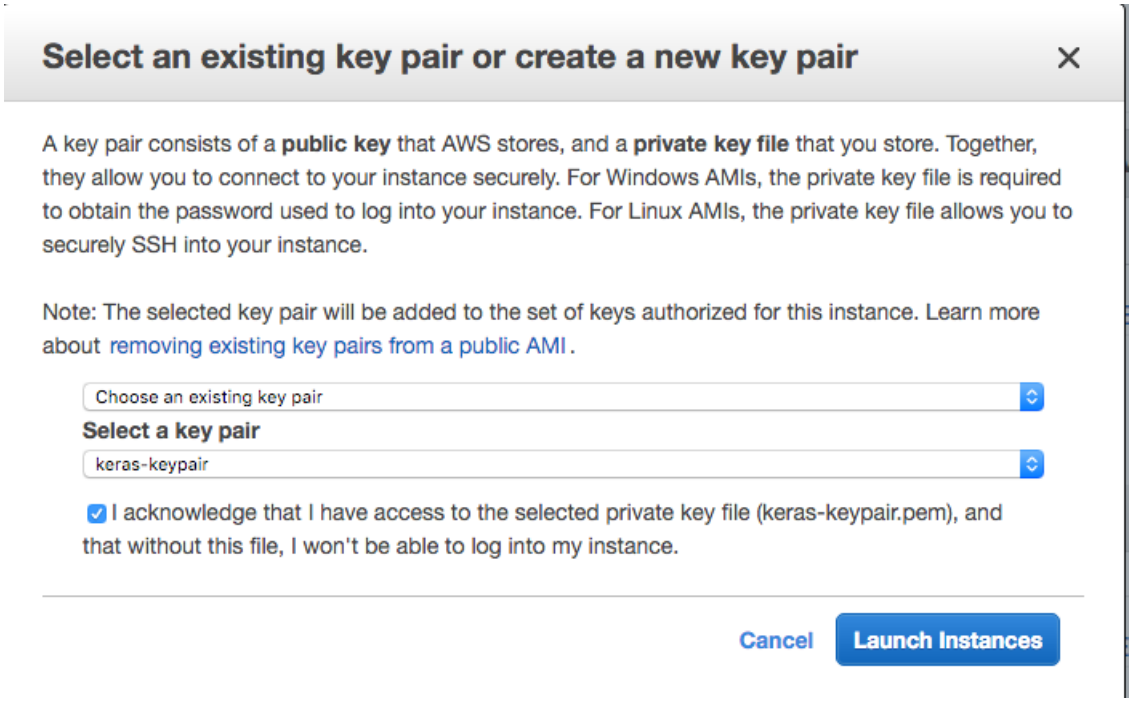

Figure C.7: Select Your Key Pair

- 12. Open a Terminal and change directory to where you downloaded your key pair.
- 13. If you have not already done so, restrict the access permissions on your key pair file. This is required as part of the SSH access to your server. For example, open a terminal on your workstation and type:

```
cd Downloads
chmod 600 keras-aws-keypair.pem
```

Listing C.1: Change Permissions of Your Key Pair File.

- 14. Click *Launch Instances*. If this is your first time using AWS, Amazon may have to validate your request and this could take up to 2 hours (often just a few minutes).
- 15. Click *View Instances* to review the status of your instance.

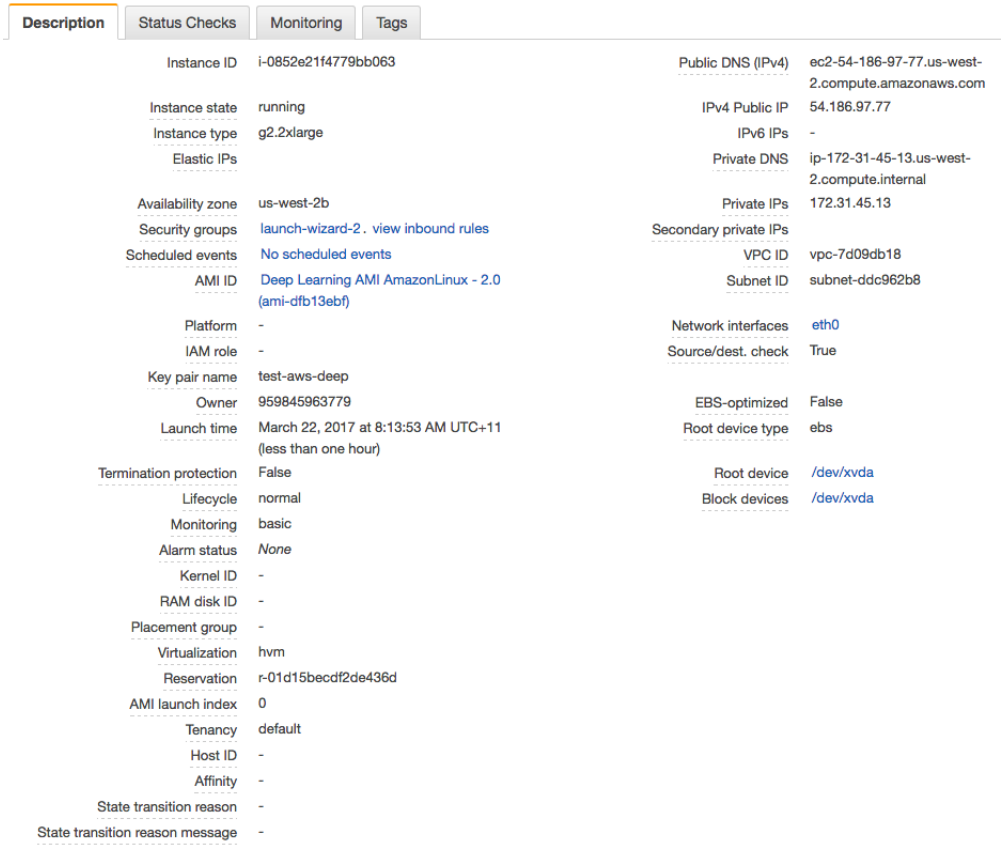

Figure C.8: Review Your Running Instance

Your server is now running and ready for you to log in.

C.4 Login, Configure and Run

Now that you have launched your server instance, it is time to log in and start using it.

- 1. Click *View Instances* in your Amazon EC2 console if you have not done so already.
- 2. Copy the *Public IP* (down the bottom of the screen in Description) to your clipboard. In this example my IP address is `54.186.97.77`. **Do not use this IP address, it will not work as your server IP address will be different**.
- 3. Open a Terminal and change directory to where you downloaded your key pair. Login to your server using SSH, for example:

```
ssh -i keras-aws-keypair.pem ec2-user@54.186.97.77
```

Listing C.2: Log-in To Your AWS Instance.

- 4. If prompted, type `yes` and press enter.

You are now logged into your server.

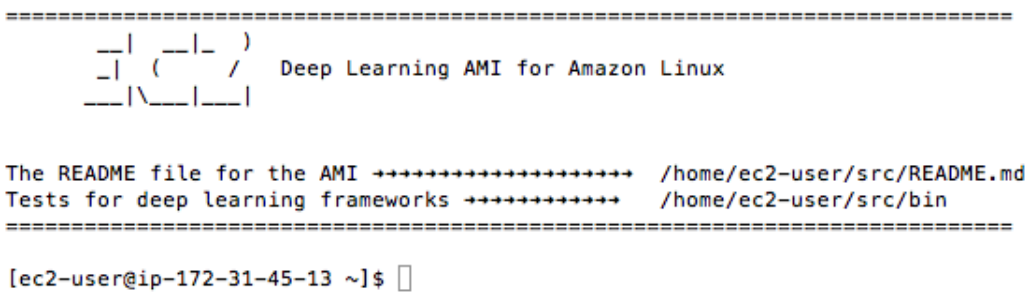

Figure C.9: Log in Screen for Your AWS Server

We need to make two small changes before we can start using Keras. This will just take a minute. You will have to do these changes each time you start the instance.

C.4.1 Update Keras

Update to a specific version of Keras known to work on this configuration, at the time of writing the latest version of Keras is version 2.0.6. We can specify this version as part of the upgrade of Keras via `pip`.

```
sudo pip install --upgrade keras==2.0.6
```

Listing C.3: Update Keras Using `pip`.

You can also confirm that Keras is installed and is working correctly by typing:

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

Listing C.4: Script To Check Keras Configuration.

You should see:

```
Using TensorFlow backend.
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcublas.so.7.5 locally
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcudnn.so.5 locally
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcufft.so.7.5 locally
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcuda.so.1 locally
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcurand.so.7.5 locally
2.0.4
```

Listing C.5: Sample Output of Script to Check Keras Configuration.

You are now free to run your code.

C.5 Build and Run Models on AWS

This section offers some tips for running your code on AWS.

C.5.1 Copy Scripts and Data to AWS

You can get started quickly by copying your files to your running AWS instance. For example, you can copy the examples provided with this book to your AWS instance using the `scp` command as follows:

```
scp -i keras-aws-keypair.pem -r src ec2-user@54.186.97.77:~/
```

Listing C.6: Example for Copying Sample Code to AWS.

This will copy the entire `src/` directory to your home directory on your AWS instance. You can easily adapt this example to get your larger datasets from your workstation onto your AWS instance. Note that Amazon may impose charges for moving very large amounts of data in and out of your AWS instance. Refer to Amazon documentation for relevant charges.

C.5.2 Run Models on AWS

You can run your scripts on your AWS instance as per normal:

```
python filename.py
```

Listing C.7: Example of Running a Python script on AWS.

You are using AWS to create large neural network models that may take hours or days to train. As such, it is a better idea to run your scripts as a background job. This allows you to close your terminal and your workstation while your AWS instance continues to run your script. You can easily run your script as a background process as follows:

```
nohup /path/to/script >/path/to/script.log 2>&1 < /dev/null &
```

Listing C.8: Run Script as a Background Process.

You can then check the status and results in your `script.log` file later.

C.6 Close Your EC2 Instance

When you are finished with your work you must close your instance. Remember you are charged by the amount of time that you use the instance. It is cheap, but you do not want to leave an instance on if you are not using it.

- 1. Log out of your instance at the terminal, for example you can type:

```
exit
```

Listing C.9: Log-out of Server Instance.

- 2. Log in to your AWS account with your web browser.
- 3. Click EC2.
- 4. Click *Instances* from the left-hand side menu.

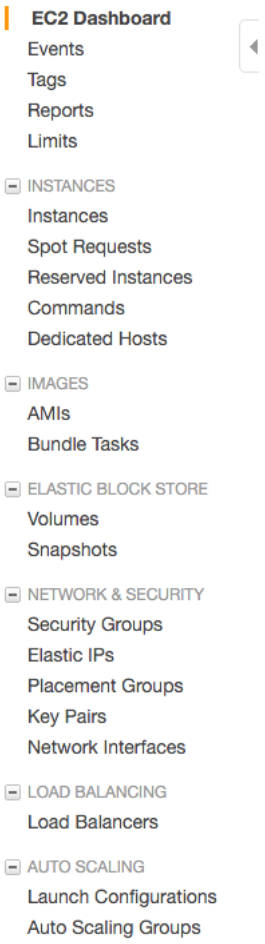

Figure C.10: Review Your List of Running Instances

- 5. Select your running instance from the list (it may already be selected if you only have one running instance).

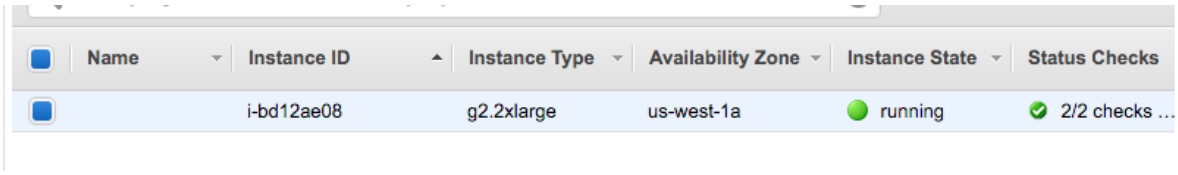

Figure C.11: Select Your Running AWS Instance

- 6. Click the *Actions* button and select *Instance State* and choose *Terminate*. Confirm that you want to terminate your running instance.

It may take a number of seconds for the instance to close and to be removed from your list of instances.

C.7 Tips and Tricks for Using Keras on AWS

Below are some tips and tricks for getting the most out of using Keras on AWS instances.

- **Design a suite of experiments to run beforehand**. Experiments can take a long time to run and you are paying for the time you use. Make time to design a batch of experiments to run on AWS. Put each in a separate file and call them in turn from another script. This will allow you to answer multiple questions from one long run, perhaps overnight.
- **Always close your instance at the end of your experiments**. You do not want to be surprised with a very large AWS bill.
- **Try spot instances for a cheaper but less reliable option**. Amazon sell unused time on their hardware at a much cheaper price, but at the cost of potentially having your instance closed at any second. If you are learning or your experiments are not critical, this might be an ideal option for you. You can access spot instances from the *Spot Instance* option on the left hand side menu in your EC2 web console.

C.8 Further Reading

Below is a list of resources to learn more about AWS and developing deep learning models in the cloud.

- An introduction to Amazon Elastic Compute Cloud (EC2) if you are new to all of this.
  http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- An introduction to Amazon Machine Images (AMI).
  http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html
- Deep Learning AMI Amazon Linux Version on the AMI Marketplace.
  https://aws.amazon.com/marketplace/pp/B01M0AXXQB

C.9 Summary

In this lesson you discovered how you can develop and evaluate your large deep learning models in Keras using GPUs on the Amazon Web Service. You learned:

- Amazon Web Services with their Elastic Compute Cloud offers an affordable way to run large deep learning models on GPU hardware.
- How to setup and launch an EC2 server for deep learning experiments.
- How to update the Keras version on the server and confirm that the system is working correctly.
- How to run Keras experiments on AWS instances in batch as background tasks.

Part VI

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

1. What LSTMs are and why they are the go-to deep learning technique for sequence prediction.
2. How LSTMs are trained using the BPTT algorithm which also imposes a way of thinking about your sequence prediction problem.
3. How to prepare data for sequence prediction including scaling, encoding values and splitting, padding and truncating input sequences.
4. How the 5-step life-cycle for LSTM models works in Keras, including define, compile, fit, evaluate, and predict.
5. How there are 4 main types of sequence prediction models and how to implement each in Keras.
6. How the vanilla LSTM is comprised of an input layer, a hidden `LSTM` layer, and a `Dense` output layer.
7. How hidden LSTM layers can be stacked but must expose the output of the entire sequence from layer to layer.
8. How CNNs can be used as input layers for LSTMs when working with image data.
9. How the Encoder-Decoder architecture can be used when predicting variable length output sequences.
10. How providing input sequences forward and backward in Bidirectional LSTMs can lift the skill on some problems.
11. How LSTMs can learn the structured relationship of input data which in turn can be used to generate new examples.
12. How to develop robust estimates of LSTM model skill and how to best start tuning model hyperparameters to get the most out of your model.
13. How a final LSTM model can be saved to file and later loaded in order to make predictions on new data.
14. How fit LSTM models can be updated when new data is made available.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable skill of being able to implement and work through sequence prediction problems using LSTMs in Python. You can now confidently bring LSTM models to your own sequence prediction problems. The sky is the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your journey with Long Short-Term Memory Recurrent Neural Networks. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2019