Long Short-Term Memory Networks With Python

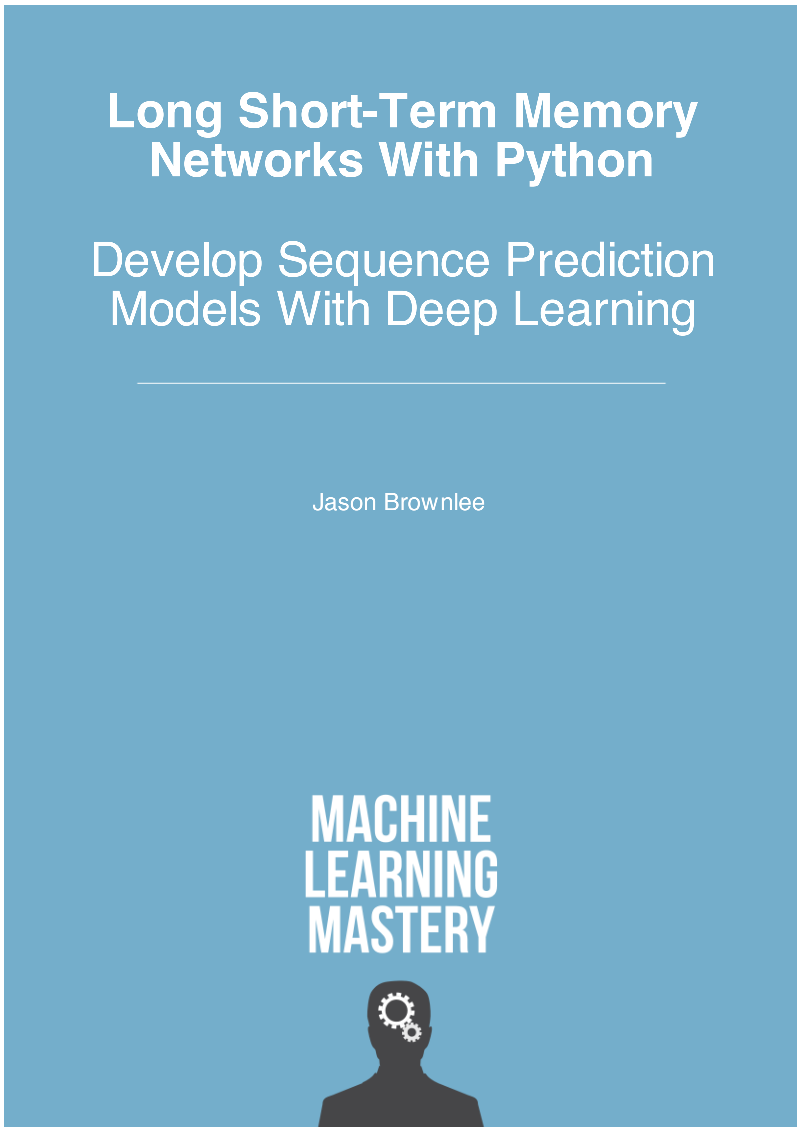
Develop Sequence Prediction Models With Deep Learning

Jason Brownlee

MACHINE
LEARNING
MASTERY

This is Just a Sample

Thank-you for your interest in **Long Short-Term Memory Networks With Python**. This is just a sample of the full text. You can purchase the complete book online from: https://machinelearningmastery.com/lstms-with-python/


Long Short-Term Memory Networks With Python
Develop Sequence Prediction Models With Deep Learning
Jason Brownlee
MACHINE LEARNING MASTERY

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
2018

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.

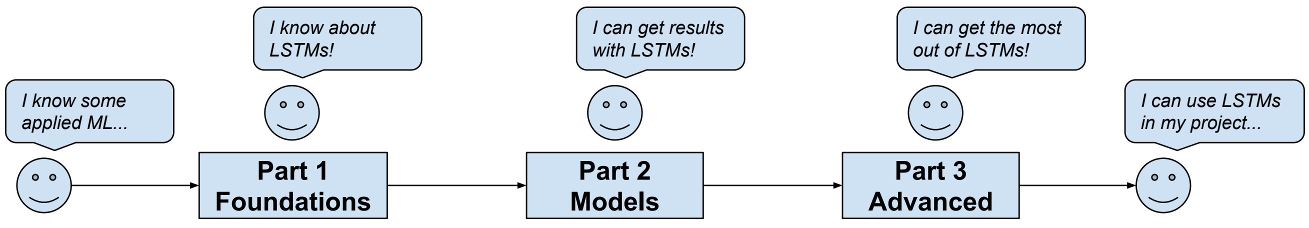

I know some applied ML...
I know about LSTMs!
Part 1 Foundations
I can get results with LSTMs!
Part 2 Models
I can get the most out of LSTMs!
Part 3 Advanced
I can use LSTMs in my project...

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, specific the different useful LSTM architectures. The focus is not the application of LSTMs to one specifically 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

Part III

Models

Chapter 1

How to Develop Vanilla LSTMs

1.0.1 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.

1.0.2 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.

1.1 The Vanilla LSTM

1.1.1 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.

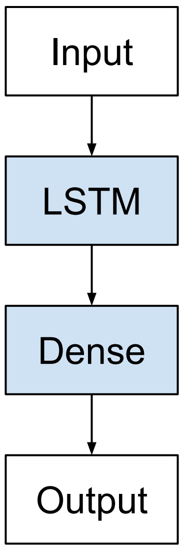


Figure 1.1: Vanilla LSTM Architecture.

1.1.2 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 1.1: 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.

1.2 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.

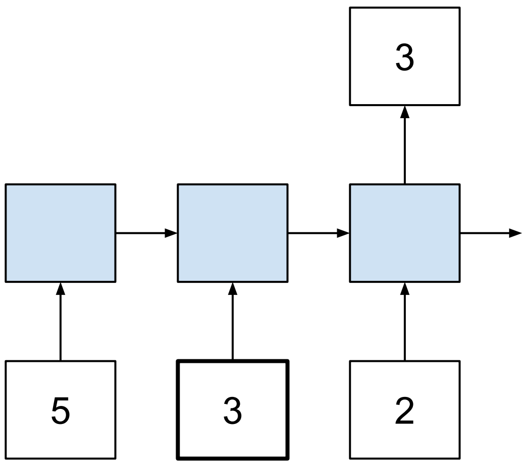


Figure 1.2: Echo sequence prediction problem framed with a many-to-one prediction model.

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.

1.2.1 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 1.2: 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 1.3: Function to generate sequences of random integers.

1.2.2 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 1.4: 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 1.5: Function to decode encoded sequences.

1.2.3 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 1.6: 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 1.7: Example output from generating sequences and encoding them.

1.2.4 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 1.8: 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 1.9: Example accessing a value in the decoded 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 1.10: 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 1.11: 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 1.12: 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.

1.3 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=['acc'])
print(model.summary())
```

Listing 1.13: 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 1.14: Example output from the defined model.

1.4 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 1.15: 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 1.16: Example output from the fitting the defined model.

1.5 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 1.17: Example of evaluating the fit LSTM model.

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

```
Accuracy: 100.000000
```

Listing 1.18: Example output from evaluating the fit model.

1.6 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 1.19: 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 1.20: 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.

1.7 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=['acc'])
print(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)*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 1.21: Example of the Vanilla LSTM applied to the Echo Problem.

1.8 Further Reading

This section provides some resources for further reading.

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

1.9 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!

1.10 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.

Part IV

Advanced

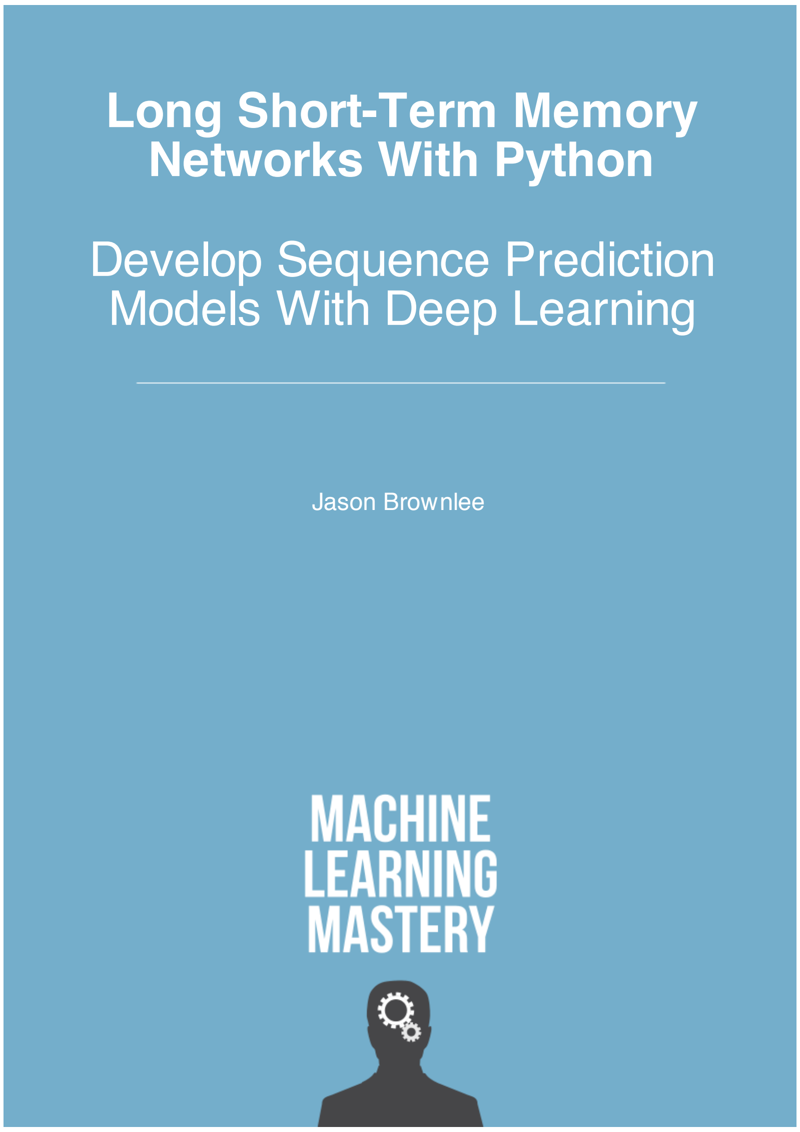
Part V

Appendix

Part VI

Conclusions

This is Just a Sample


Long Short-Term Memory Networks With Python
Develop Sequence Prediction Models With Deep Learning
Jason Brownlee
MACHINE LEARNING MASTERY