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 involves 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.

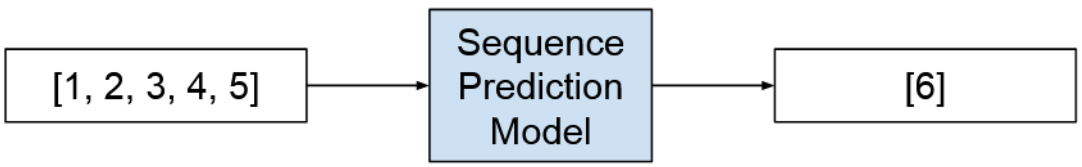

Figure 1.1: Depiction of a sequence prediction problem.

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.

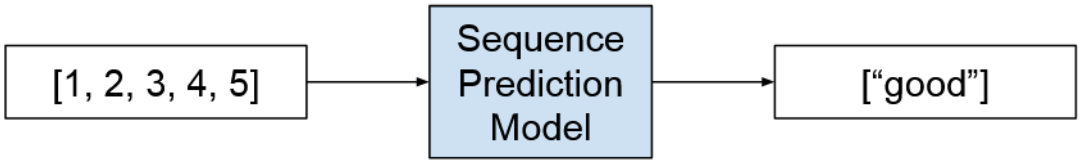

Figure 1.2: Depiction of a sequence classification problem.

> 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.

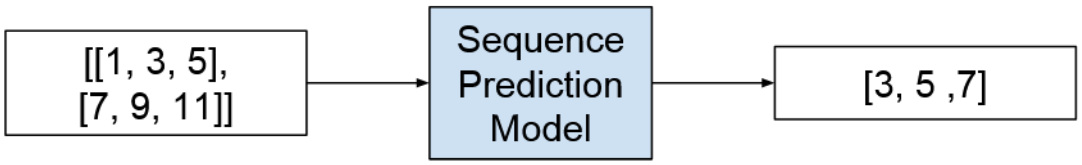

Figure 1.3: Depiction of a sequence generation problem.

> [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.

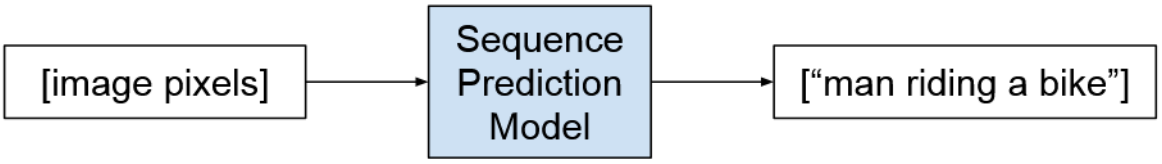

Figure 1.4: Depiction of a sequence generation problem for captioning an image.

> 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.

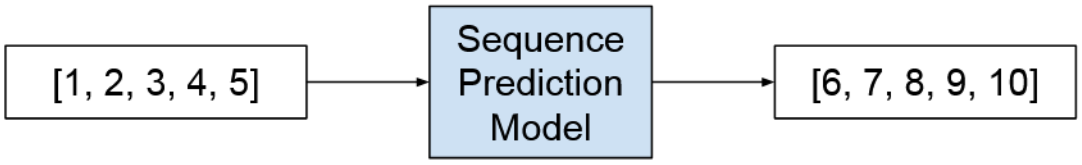

Figure 1.5: Depiction of a sequence-to-sequence prediction problem.

> 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 arbitrary 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 subfeld 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 latterly (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 time series 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 ow 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.

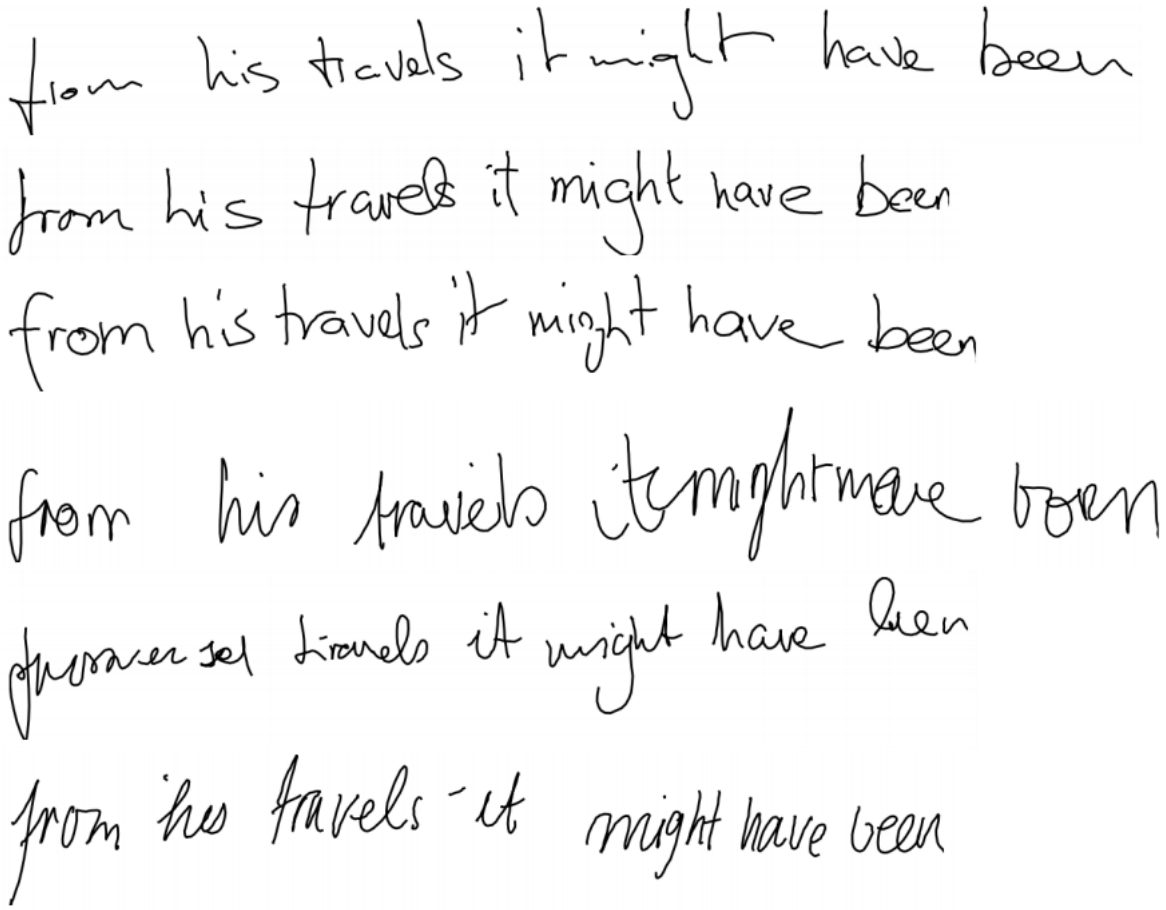

Figure 1.8: Example of LSTM generated captions, taken from *Generating Sequences With Recurrent Neural Networks*, 2014.

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

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)
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

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.

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 testing. 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