Chapter 2

How to Train LSTMs

2.0.1 Lesson Goal

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

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

2.0.2 Lesson Overview

This lesson is divided into 6 parts; they are:

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

Let's get started.

2.1 Backpropagation Training Algorithm

Backpropagation refers to two things:

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

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

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

2.2 Unrolling Recurrent Neural Networks

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

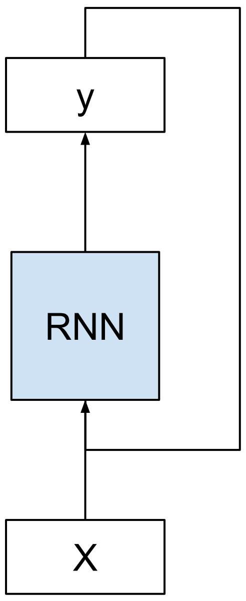

Figure 2.1: Example of a simple recurrent neural network.

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

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

2.2.1 Unfolding the Forward Pass

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

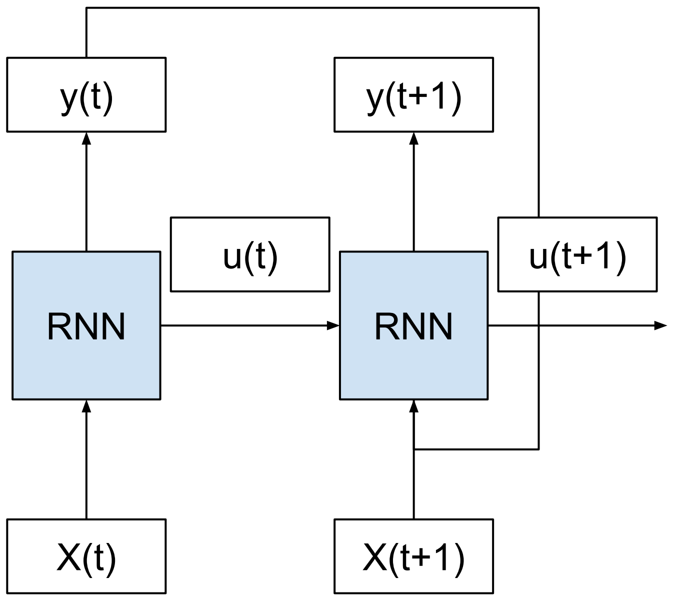

Figure 2.2: Example of an unfolded recurrent neural network.

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

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

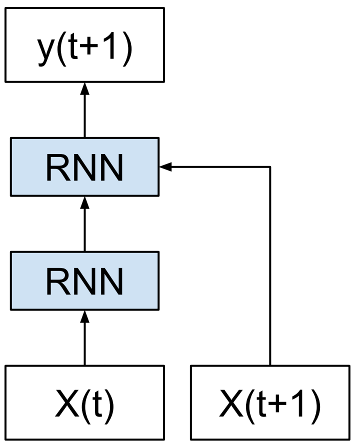

Figure 2.3: Example of an unfolded recurrent neural network with layers.

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

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

2.2.2 Unfolding the Backward Pass

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

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

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

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

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

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

— *Online Sequence Training of Recurrent Neural Networks with Connectionist Temporal Classification*, 2015.

2.3 Backpropagation Through Time

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

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

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

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

> One of the main problems of BPTT is the high cost of a single parameter update, which makes it impossible to use a large number of iterations.

— *Training Recurrent Neural Networks*, 2013

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

2.4 Truncated Backpropagation Through Time

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

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

We can summarize the algorithm as follows:

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

The TBPTT algorithm requires the consideration of two parameters:

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

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

2.5 Configurations for Truncated BPTT

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

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

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

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

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

2.6 Keras Implementation of TBPTT

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

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

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

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

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

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

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

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

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

2.7 Further Reading

This section provides some resources for further reading.

2.7.1 Books

- *Neural Smithing*, 1999.
  http://amzn.to/2u9yjJh

- *Deep Learning*, 2016.
  http://amzn.to/2sx7oFo

- *Supervised Sequence Labelling with Recurrent Neural Networks*, 2008.
  http://amzn.to/2upsSJ9

2.7.2 Research Papers

- *Online Sequence Training of Recurrent Neural Networks with Connectionist Temporal Classification*, 2015.
  https://arxiv.org/abs/1511.06841

- *Framewise phoneme classification with bidirectional LSTM and other neural network architectures*, 2005.

- *Deep learning*, Nature, 2015.

- *Training Recurrent Neural Networks*, 2013.

- *Learning Representations By Backpropagating Errors*, 1986.

- *Backpropagation Through Time: What It Does And How To Do It*, 1990.

- *An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories*, 1990.

- *Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity*, 1995.

2.8 Extensions

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

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

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

2.9 Summary

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

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

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

Chapter 3

How to Prepare Data for LSTMs

3.0.1 Lesson Goal

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

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

3.0.2 Lesson Overview

This lesson is divided into 4 parts; they are:

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

Let's get started.

3.1 Prepare Numeric Data

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

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

3.1.1 Normalize Series Data

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

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

- **Fit the scaler using available training data**. For normalization, this means the training data will be used to estimate the minimum and maximum observable values. This is done by calling the `fit()` function.

- **Apply the scale to training data**. This means you can use the normalized data to train your model. This is done by calling the `transform()` function.

- **Apply the scale to data going forward**. This means you can prepare new data in the future on which you want to make predictions.

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

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

Listing 3.1: Example of normalizing a sequence.

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

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

Min: 10.000000, Max: 100.000000

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

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

Listing 3.2: Example output from normalizing a sequence.

3.1.2 Standardize Series Data

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

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

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

```python
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:

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

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

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

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

Listing 3.20: Example output of the created series.

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

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

Listing 3.21: Example of shifting the series forward.

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

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

Listing 3.22: Example output of shifting the series forward.

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

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

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

Listing 3.23: Example of shifting the series backward.

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

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

Listing 3.24: Example output of shifting the series backward.

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

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

3.5 Further Reading

This section provides some resources for further reading.

3.5.1 Numeric Scaling APIs

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

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

3.5.2 Categorical Encoding APIs

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

3.5.3 Varied Length Sequence APIs

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

3.5.4 Supervised Learning APIs

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

3.6 Extensions

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

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

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

3.7 Summary

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

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

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

Chapter 4

How to Develop LSTMs in Keras

4.0.1 Lesson Goal

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

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

4.0.2 Lesson Overview

This lesson is divided into 7 parts; they are:

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

Let's get started.

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

4.1 Define the Model

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

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

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

Listing 4.1: Example of defining an LSTM model.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2 Compile the Model

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

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

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

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

Listing 4.7: Example of compiling an LSTM model.

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

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

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

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

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

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

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

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

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

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

4.3 Fit the Model

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

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

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

Below are some common configurations for the batch size:

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

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

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

Listing 4.10: Example of fitting an LSTM model.

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

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

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

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

4.4 Evaluate the Model

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

4.9 Extensions

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

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

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

4.10 Summary

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

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

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

Chapter 5

Models for Sequence Prediction

5.0.1 Lesson Goal

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

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

5.0.2 Lesson Overview

This lesson is divided into 5 parts; they are:

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

Let's get started.

5.1 Sequence Prediction

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

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

Listing 5.1: Example of the general LSTM model.

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

5.2 Models for Sequence Prediction

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

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

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

5.2.1 One-to-One Model

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

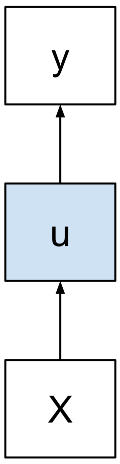

Figure 5.1: One-to-One Sequence Prediction Model.

For example:

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

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

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

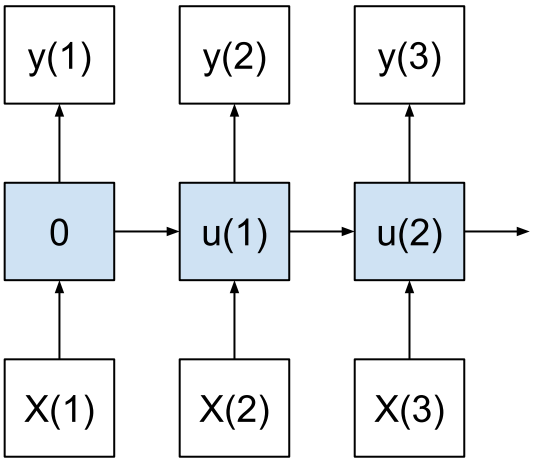

Figure 5.2: One-to-One Sequence Prediction Model Over Time.

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

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

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

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

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

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

5.2.2 One-to-Many Model

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

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

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

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

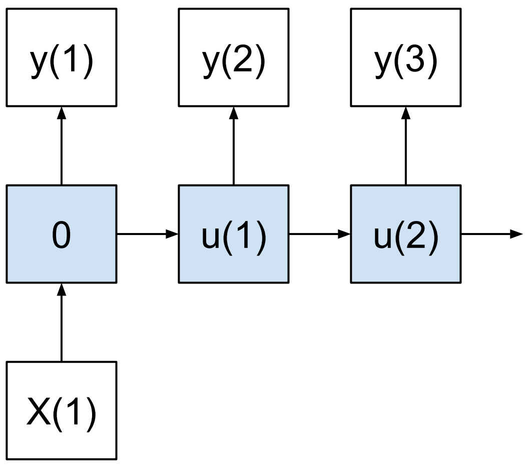

Figure 5.3: One-to-Many Sequence Prediction Model.

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

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

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

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

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

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

5.2.3 Many-to-One Model

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

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

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

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

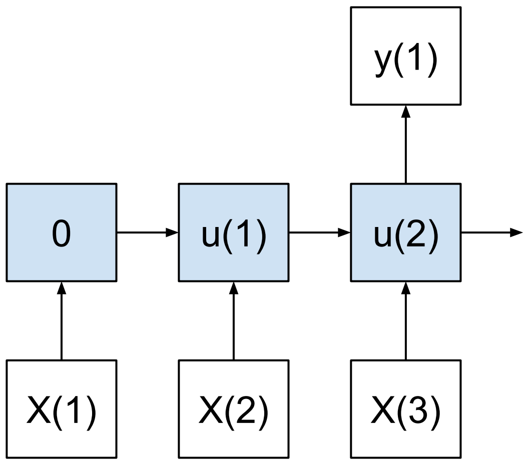

Figure 5.4: Many-to-One Sequence Prediction Model.

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

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

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

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

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

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

5.2.4 Many-to-Many Model

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

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

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

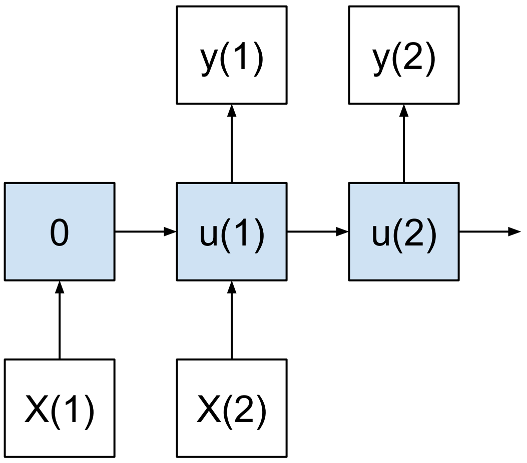

Figure 5.5: Many-to-Many Sequence Prediction Model.

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

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

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

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

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

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

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

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

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

5.3 Mapping Applications to Models

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

5.3.1 Time Series

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

5.3.2 Natural Language Processing

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

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

5.4 Cardinality from Time Steps (not Features!)

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

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

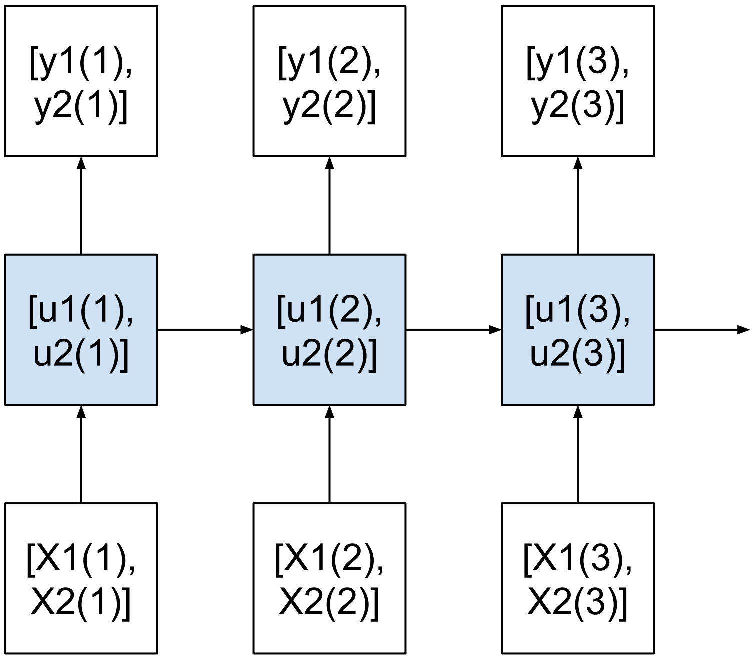

Figure 5.6: Multiple-Feature Sequence Prediction Model.

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

5.5 Two Common Misunderstandings

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

5.5.1 Time steps as Input Features

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

5.5.2 Time steps as Output Features

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

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

5.6 Further Reading

This section provides some resources for further reading.

5.6.1 Articles

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

5.7 Extensions

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

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

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

5.8 Summary

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

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

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