1
Recurrent Neural Networks

In the last chapter, we covered the ability of **Convolutional Neural Networks** (**CNNs**) to learn feature representations from grid-like data. In this chapter, we introduce **Recurrent Neural Networks** (**RNNs**), which are designed for processing sequential data.

Feedforward Neural Networks (**FFNNs**) treat the feature vectors for each sample as independent and identically distributed. As a result, they do not systematically take prior data points into account when evaluating the current observation. In other words, they have no memory.

One-dimensional convolutions, which we covered in the previous chapter, produce sequence elements that are a function of a small number of their neighbors. However, they only allow for shallow parameter-sharing by applying the same convolutional kernel to the relevant time steps.

The major innovation of the RNN model is that each output is a function of the previous output and new information, so that RNNs gain the ability to incorporate information on previous observations into the computation it performs on a new feature vector.

We will illustrate how this recurrent formulation enables parameter-sharing across a much deeper computational graph, which now includes cycles. You will also encounter **Long Short-Term Memory** (**LSTM**) units and **Gated Recurrent Units** (**GRUs**), which aim to overcome the challenge of vanishing gradients associated with learning long-range dependencies, where errors need to be propagated over many connections.

RNNs are successfully applied to various tasks that require mapping one or more input sequences to one or more output sequences, and are particularly well-suited to natural language. We will explore how RNNs can be applied to univariate and multivariate time series to predict market or fundamental data. We will also cover how RNNs can be applied to alternative text using word embeddings, which we covered in `Chapter 15`, *Word Embeddings*, to classify the sentiment expressed in documents.

Most specifically, in this chapter, you will learn about the following:

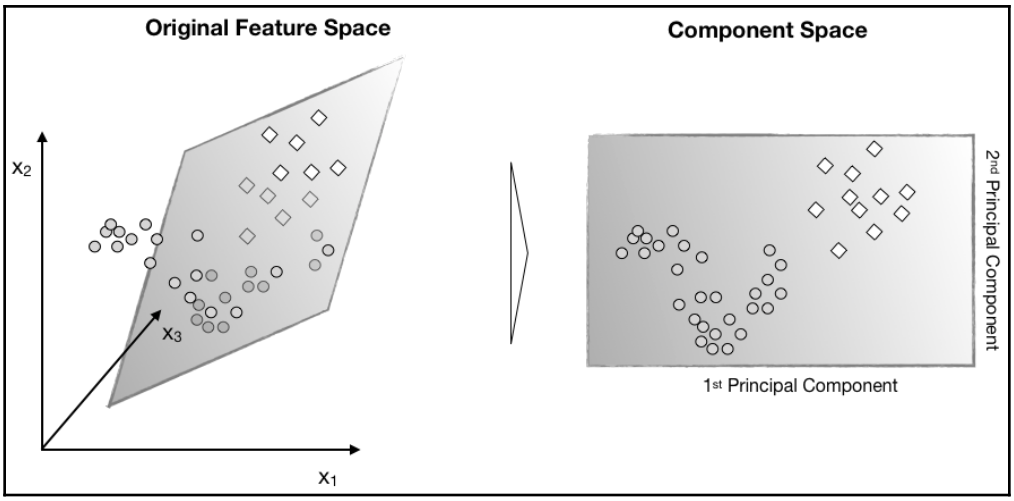
- How to unroll and analyze the computational graph for RNNs
- How gated units learn to regulate RNN memory from data to enable long-range dependencies
- How to design and train RNNs for univariate and multivariate time series in Python
- How to leverage word embeddings for sentiment analysis with RNNs

How RNNs work

RNNs assume that data is sequential so that previous data points impact the current observation and are relevant for predictions of subsequent elements in the sequence.

They allow for more complex and diverse input-output relationships than FFNNs and CNNs, which are designed to map one input to one output vector, usually of a fixed size, and using a given number of computational steps. RNNs, in contrast, can model data for tasks where the input, the output, or both, are best represented as a sequence of vectors.

The following diagram, inspired by Andrew Karpathy's 2015 blog post on the *Unreasonable Effectiveness of RNN* (you can refer to GitHub for more information, here: `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`), illustrates mappings from input to output vectors using non-linear transformations carried out by one or more neural network layers:



The left panel shows one-to-one mapping between vectors of fixed sizes, that is, for example, typical for FFNNs and CNNs, for image classification covered in the last chapter.

The other three panels show various RNN applications that map the input to the output vectors by applying a recurrent transformation to the new input and the state produced by the previous iteration. The xinput vectors to an RNN are also called **context**.

The vectors are time-indexed, which is a usual requirement of trading-related applications, but they could also be labeled by a different set of sequential values. Generic sequence-to-sequence mapping tasks and sample applications can include the following:

- **One-to-many**: Image captioning, for example, takes a single pixel vector (as mentioned in the last chapter) and maps it to a sequence of words.
- **Many-to-one**: Sentiment analysis takes a sequence of words or tokens (refer to `Chapter 13`, *Working with Text Data*) and maps it to an output scalar or vector.
- **Many-to-many**: Machine translations, or the labeling of video frames, map sequences of input vectors to sequences of output vectors, either in a synchronized (as shown), or in an asynchronous fashion. Multi-step prediction of multivariate time series also maps several input vectors to several output vectors.

Note that input and output sequences can be of arbitrary lengths because the recurrent transformation, which is fixed but learned from the data, can be applied as many times as needed.

Just as CNNs can easily scale to large images, with some CNNs processing images of variable sizes, RNNs scale to much longer sequences than networks that are not tailored to sequence-based tasks. Most RNNs can also process sequences of variable length.

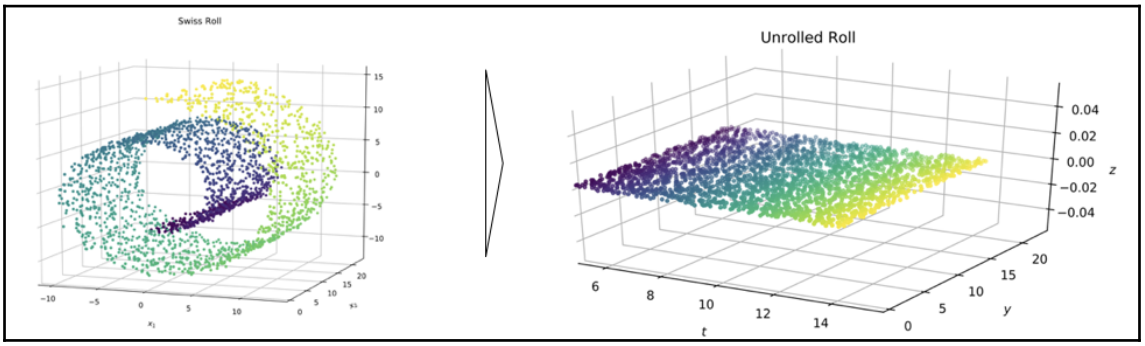
Unfolding a computational graph with cycles

As suggested in the previous section, RNNs are called **recurrent** because they apply the same transformations to every element of a sequence in such a way that the output depends on the outcome of prior iterations. As a result, RNNs maintain an internal state that captures information about previous elements in the sequence, just like memory.

The following diagram shows the computational graph implied by a simple hidden RNN unit learning two weight matrices during its training:

- $\mathbf{W_{hh}}$: Applies to the previous hidden state, h_{t-1}
- $\mathbf{W_{hx}}$: Applies to the current input, x_t

A non-linear transformation of the sum of the two matrix multiplications—for example, using the tanh or ReLU activation functions—becomes the RNN layer's output, y_t:



$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

The right side of the equation shows the effect of unrolling the recurrent relationship, highlighting the repeated application of the same transformations and the resulting state that combines information from past sequence elements with the current input, or context. An alternative formulation connects the context vector to the first hidden state only; we will outline additional options to modify the previously shown baseline architecture in the following section.

The recurrent connections between subsequent hidden states are critical in rendering the RNN model as universal because it can compute any (discrete) functions that a Turing machine can represent.

Backpropagation through time

The unrolled computational graph shown in the preceding diagram highlights that the learning process necessarily encompasses all time steps included in a given input sequence. The backpropagation algorithm, which updates the weight parameters based on the gradient of the loss function with respect to the parameters, involves a forward pass from left to right along the unrolled computational graph, followed by a backward pass in the opposite direction.

Just like the backpropagation techniques discussed in `Chapter 17`, *Deep Learning*, the algorithm evaluates a loss function to obtain the gradients and update the weight parameters. In the RNN context, backpropagation runs from right to left in the computational graph, updating the parameters from the final time step all the way to the initial time step. Hence, the algorithm is called **backpropagation through time**.

It highlights both the power of RNNs to model long-range dependencies by sharing parameters across an arbitrary number of sequence elements and maintaining a corresponding state. On the other hand, it is computationally quite expensive, and the computations for each time step cannot be parallelized due to its inherently sequential nature.

Alternative RNN architectures

Just like the feedforward and CNN architectures we covered in the previous two chapters, RNNs can be designed in a variety of ways that best capture the functional relationship and dynamic between input and output data.

In addition to the recurrent connections between the hidden states, there are several alternative approaches, including recurrent output relationships, bidirectional RNNs, and encoder-decoder architectures (you can refer to GitHub for more background references to complement this brief summary here: `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`).

Output recurrence and teacher forcing

One way to reduce the computational complexity of hidden state recurrences is to connect a unit's hidden state to the prior unit's output rather than its hidden state. The resulting RNN has a lower capacity than the architecture discussed previously, but different time steps can be trained are now decoupled and can be trained in parallel.

However, the successful learning of relevant past information requires the training output samples to capture this information so that backpropagation can adjust the network parameters accordingly. The use of previous outcome values alongside the input vectors is called **teacher forcing**.

Connections from the output to the subsequent hidden states can be also used in combination with hidden recurrences. However, training requires backpropagation through time and cannot be run in parallel.

Bidirectional RNNs

For some tasks, it can be realistic, necessary, and beneficial for the output to depend not only on past sequence elements as modeled so far, but also on future elements. Machine translation or speech and handwriting recognition are examples where subsequent sequence elements are useful and realistically available to disambiguate competing outputs.

For a one-dimensional sequence, bidirectional RNNs combine an RNN model that moves forward with another RNN that scans the sequence in the opposite direction. As a result, the output comes to depend on both the future and the past of the sequence. Applications to these domains have been very successful.

Bidirectional RNNs can also be used with two-dimensional image data. In this case, one pair of RNNs performs the forward and backward processing of the sequence in each dimension.

Encoder-decoder architectures and the attention mechanism

The architectures discussed so far assumed that the input and output sequences have equal length. Encoder-decoder architectures, also called **sequence-to-sequence** (**seq2seq**) architectures, relax this assumption and have become very popular for machine translation and seq2seq prediction in general.

The encoder is an RNN model that maps the input space to a different space, also called the **latent space**, whereas the decoder function is a complementary RNN model that maps the encoded input to the target space. In the next chapter, we will cover autoencoders, which are able to learn a feature representation in an unsupervised setting using a variety of deep learning architectures.

Encoder-decoder RNNs are trained jointly so that the input of the final encoder's hidden state becomes the input to the decoder's hidden state, which in turn learns to match the training samples.

The attention mechanism addresses a limitation of using fixed-size encoder inputs when input sequences themselves vary in size. The mechanism converts raw text data into a distributed representation (see `Chapter 15`, *Word Embeddings*), stores the result, and uses a weighted average of these feature vectors as context.

A recent transformer architecture dispenses with recurrence and convolutions and exclusively relies on this attention mechanism to learn input-output mappings. It has achieved superior quality on machine translation tasks while requiring much less time for training, not least because it can be parallelized.

How to design deep RNNs

The unrolled computational graph in the previous diagram shows that each transformation, consisting of a linear matrix operation followed by a non-linear transformation, is shallow in the sense that it could be represented by a single network layer.

Just as added depth resulted in more useful hierarchical representations learned by FFNNs, and in particular CNNs, RNNs can also benefit from decomposing the input-output mapping into multiple layers. In the RNN context, this mapping consists of transformations between the input and hidden state, between hidden states, and from the hidden state to the output.

A common approach that we will employ next is to stack recurrent layers on top of each other to have these layers learn a hierarchical temporal representation of the input data. This means that a lower layer may capture higher-frequency patterns, synthesized by a higher layer into lower-frequency characteristics that prove useful for the classification or regression task.

Less popular alternatives include adding layers to the connections from input to hidden, between hidden states, or from hidden to output, possibly using skip connections to avoid the shortest path between time steps increasing and optimization becoming more difficult.

The challenge of learning long-range dependencies

In theory, RNNs can make use of information in arbitrarily long sequences. However, in practice, they are limited to looking back only a few steps.

More specifically, RNNs struggle to derive useful context information from time steps that are far from the current observation. The fundamental problem is the impact on gradients of propagation over many time steps. Due to repeated multiplication, the gradients tend to either vanish, that is, decrease toward zero (the typical case), or explode, that is, grow toward infinity (this happens less frequently, but renders optimization very difficult).

Even if the parameters are such that stability is not at risk and the network is able to store memories, long-term interactions will receive exponentially smaller weights due to the multiplication of many Jacobian matrices containing the gradient information.

Experiments have shown that successful RNN training using the **Stochastic Gradient Descent** (**SGD**) method is quite challenging with very little chance of success for sequences with only 10 or 20 elements.

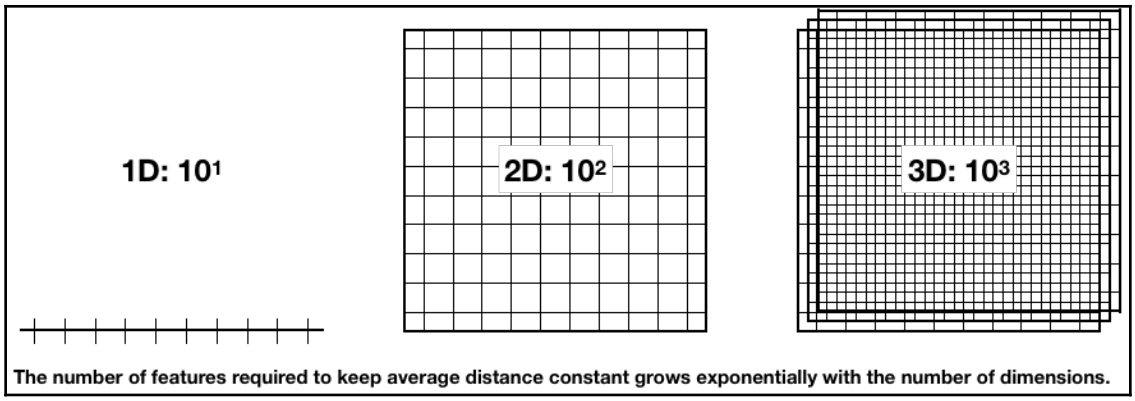
Several RNN design techniques have been introduced to address this challenge, such as echo state networks and leaky units. The latter operates at different time scales, focusing part of the model on higher-frequency and other parts on lower-frequency representations to deliberately learn and combine different aspects from the data. Strategies include connections that skip time steps or units that integrate signals from different frequencies. For additional background information from GitHub, refer to `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`.

The most successful approach to learning dependencies over hundreds of time steps uses gated units, which are trained to regulate how much past information a unit maintains in its current state and when to reset or forget this information. The most popular examples include LSTM units and GRUs.

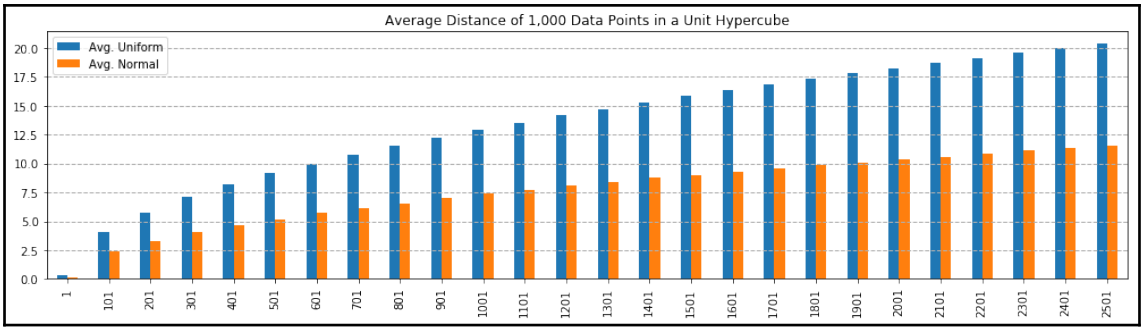
LSTM architectures

RNNs with an LSTM architecture have more complex units that maintain an internal state and contain gates to keep track of dependencies between elements of the input sequence and regulate the cell states accordingly. These gates recurrently connect to each other instead of the usual hidden units we have previously encountered. They aim to address the problem of vanishing and exploding gradients by letting gradients pass through unchanged.

The following diagram shows unrolled LSTM units and outlines their typical gating mechanism (for more information from GitHub, refer to `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`):



A typical LSTM unit combines four parameterized layers that interact with each other and the cell state by transforming and passing along vectors. These layers usually involve an input gate, an output gate, and a forget gate, but there are variations that may have additional gates or lack some of these mechanisms. The yellow nodes in the following diagram identify element-wise operations and the gray elements represent layers with weight and bias parameters learned during training:



The **cell state**, *c*, passes long the horizontal connection at the top of the cell. The cell state's interaction with the various gates leads to a series of recurrent decisions, as follows:

1. The **forget gate** controls how much of the cells state should be voided. It receives the prior hidden state, h_{t-1}, and the current input, x_t, as inputs, computes a sigmoid activation and multiplies the resulting value, f_t, in [0, 1] with the cell state, reducing or keeping it accordingly.

2. The input **gate** also computes a sigmoid activation, i_t, from h_{t-1} and x_t, which produce update candidates. A tanh activation in the range of [-1, 1] multiplies the update candidates, u_t, and, depending on the resulting sign, adds or subtracts the result from the cell state.
3. The **output gate** filters the updated cell state using a sigmoid activation, o_t, and multiplies it by the cell state normalized to the range [-1, 1] using a tanh activation.

GRUs

GRUs simplify LSTM units by omitting the output gate. They have been shown to achieve a similar performance on certain language modeling tasks but do better on smaller datasets.

GRUs aim at making each recurrent unit adapt to capturing dependencies of different time scales. Similarly to the LSTM unit, GRUs have gating units that modulate the flow of information inside the unit but discard separate memory cells (for additional details from GitHub, refer to `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`).

How to build and train RNNs using Python

In this section, we will illustrate how to build RNNs using the Keras library for various scenarios. The first set of models includes regression and classification of univariate and multivariate time series. The second set of tasks focuses on text data for sentiment analysis using text data converted to word embeddings (see `Chapter 15`, *Word Embeddings*).

More specifically, we'll first demonstrate how to prepare time series data to predict the next value for univariate time series with a single LSTM layer to predict stock index values.

Next, we will add depth and use stacked LSTM layers combined with learned embeddings and one-hot-encoded categorical data to build an RNN model with three distinct inputs that classifies asset price movements. Finally, we will demonstrate how to model multivariate time series using RNNs.

In the second part, we will return to Yelp business reviews and use word vectors learned in `Chapter 15`, *Word Embeddings*, to classify the sentiment expressed in the reviews.

Univariate time series regression

In this subsection, we will forecast the S&P 500 Index values (see the `univariate_time_series_regression` notebook for implementation details).

We obtain data for 2010-2018 from the **Federal Reserved Bank's Data Service** (**FRED**—see `Chapter 2`, *Market and Fundamental Data*), as follows:

```
sp500 = web.DataReader('SP500', 'fred', start='2010',
                       end='2019').dropna()
sp500.info()
DatetimeIndex: 2264 entries, 2010-01-04 to 2018-12-31
Data columns (total 1 columns):
     SP500 2264 non-null float64
```

We process the data by scaling it to the [0, 1] interval using scikit-learn's `minmax_scale` function, as shown here:

```
from sklearn.preprocessing import minmax_scale
sp500_scaled = sp500.apply(minmax_scale)
```

How to get time series data into shape for a RNN

We will generate sequences of 63 trading days, approximately three months, and use a single LSTM layer with 20 hidden units to predict the index value one time step ahead.

The input to every LSTM layer must have three dimensions:

- **Samples**: One sequence is one sample. A batch contains one or more samples.
- **Time steps**: One time step is one point of observation in the sample.
- **Features**: One feature is one observation at a time step.

Our S&P 500 sample has 2,264 observations or time steps. We will create overlapping sequences using a window of 63 observations each.

For a simpler window of size T = 5, we obtain input-output pairs as shown in the following code snippet:

```
$$\begin{ar$$y}{c|c}
\text{Input} & \text{Output}\\
\hline {\langle x_1,x_2,x_3,x_4,x_5\rangle} & { x_6} \\
\ {\langle x_{2},x_{3},x_{4},x_{5},x_{6} \rangle } & {x_{7} } \\
{\vdots} & {\vdots}\\
{ \langle x_{T-5},x_{T-4},x_{T-3},x_{T-2},x_{T-1} \rangle } & {x_{T}}
\end{array}$$
```

We can use the `create_univariate_rnn_data()` function to stack the sequences selected using a rolling window, as follows:

```
defcreate_univariate_rnn_data(data, window_size):
y = data[window_size:]
data = data.values.reshape(-1, 1) # make 2D
n = data.shape[0]
X = np.hstack(tuple([data[i: n-j, :] for i, j in
enumerate(range(window_size, 0, -1))]))
return pd.DataFrame(X, index=y.index), y
```

We apply this function to the rescaled stock index for a `window_size=63` function to obtain a two-dimensional dataset of shape number of samples `X` number of time steps, as described here:

```
X, y = create_univariate_rnn_data(sp500_scaled, window_size=63)
X.shape
(2201, 63)
```

We will use data from 2018 as a test set and reshape the features to the requisite three-dimensional format by adding a single third-dimension, as follows:

```
X_train = X[:'2017'].values.reshape(-1, window_size, 1)
y_train = y[:'2017']
# keep the last year for testing
X_test = X['2018'].values.reshape(-1, window_size, 1)
y_test = y['2018']
```

How to define a two-layer RNN using a single LSTM layer

Having created input/output pairs from our stock index time series and split the pairs into training/testing sets, we can now begin setting up our RNN. Keras makes it very straightforward to build a two-hidden-layer RNN of the following specifications:

- Layer 1 uses an LSTM module with 20 hidden units (note here the bit that reads `input_shape = (window_size,1)`).
- Layer 2 uses a fully connected module with one unit.
- We use the `mean_squared_error` loss because we are performing regression.

This can be constructed using just a few lines, as follows:

```
rnn = Sequential([
LSTM(units=20,
input_shape=(window_size, n_features), name='LSTM'),
Dense(1, name='Output')
])
```

The summary shows that the model has 1,781 parameters:

```
rnn.summary()
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
LSTM (LSTM) (None, 20) 1760
_________________________________________________________________
Output (Dense) (None, 1) 21
=================================================================
Total params: 1,781
Trainable params: 1,781
Non-trainable params: 0
```

We train the model using the `RMSProp` optimizer recommended for RNNs with default settings and compile the model with **mean squared error** (**MSE**) for this regression problem, as indicated here:

```
optimizer = keras.optimizers.RMSprop(lr=0.001,
                                     rho=0.9,
                                     epsilon=1e-08,
                                     decay=0.0)
rnn.compile(loss='mean_squared_error', optimizer=optimizer)
```

We define an `EarlyStopping` callback function and train the model for 500 episodes, as follows:

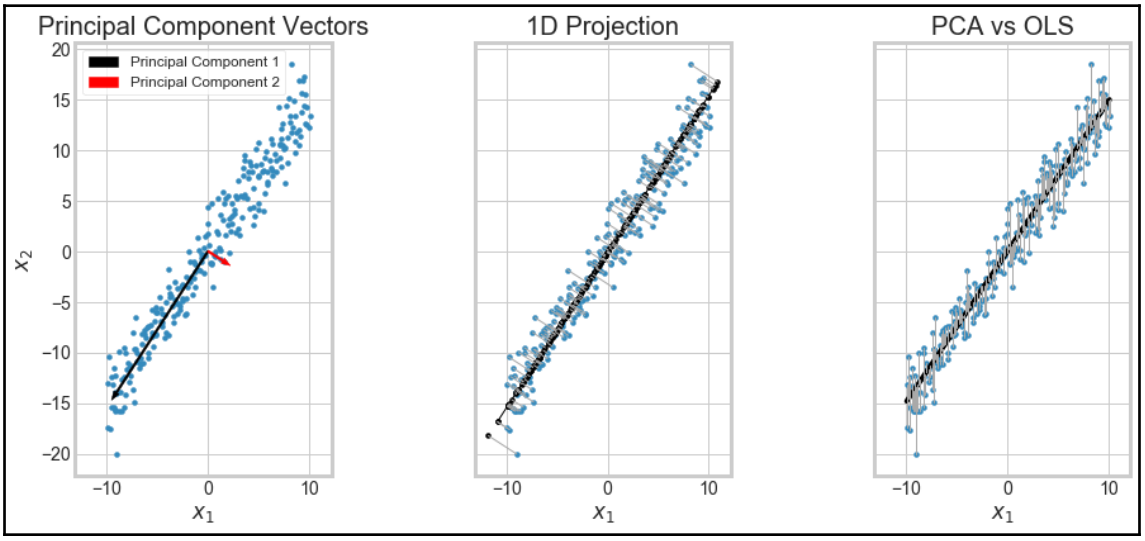
```
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=25,
                               restore_best_weights=True)
result = rnn.fit(X_train,
                 y_train,
                 epochs=500,
                 batch_size=20,
                 validation_data=(X_test, y_test),
                 callbacks=[checkpointer, early_stopping],
                 verbose=1)
```

Training stops after 136 epochs and we reload the weights for the best model:

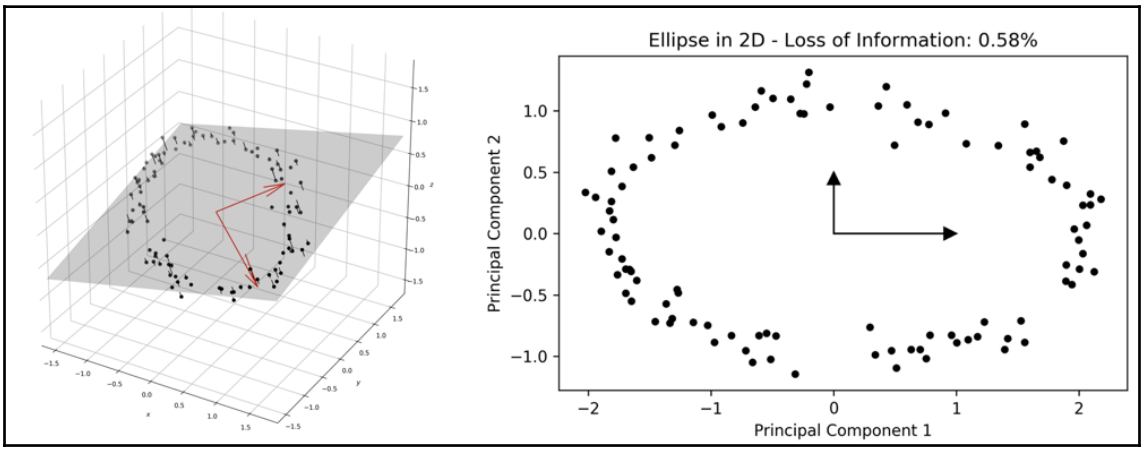
```
rnn.load_weights(rnn_path)
```

The `loss_history` function shows how the model's validation error converges to an error level that illustrates the noise inherent in predicting stock prices, as follows:

```
loss_history = pd.DataFrame(result.history).pow(.5)
loss_history.columns=['Training RMSE', 'Validation RMSE']
loss_history.plot(logy=True, lw=2);
```



The following diagrams illustrate the out-of-sample forecast performance that generally tracks the 2018 index development with a test RMSE of 0.015 on the rescaled price series. The test IC is 95.85%, as shown here:



Stacked LSTMs for time series classification

We'll now build a slightly deeper model by stacking two LSTM layers using the Quandl stock price data (see the `stacked_lstm_with_feature_embeddings` notebook for implementation details). Furthermore, we will include features that are not sequential in nature, namely indicator variables for identifying the equity and the month.

How to prepare the data

We load the Quandl adjusted stock price data (see the instructions on GitHub on how to obtain the source data) as follows (see the `build_dataset` notebook):

```
prices = (pd.read_hdf('../data/assets.h5', 'quandl/wiki/prices')
.adj_close
.unstack().loc['2007':])
prices.info()
DatetimeIndex: 2896 entries, 2007-01-01 to 2018-03-27
Columns: 3199 entries, A to ZUMZ
```

We start by generating weekly returns for close to 2,500 stocks without missing data for the 2008-17 period, as follows:

```
returns = (prices
.resample('W')
.last()
.pct_change()
.loc['2008': '2017']
.dropna(axis=1)
.sort_index(ascending=False))
returns.info()
DatetimeIndex: 2576 entries, 2017-12-29 to 2008-01-01
Columns: 2489 entries, A to ZUMZ
```

We'll use 52-week sequences, which we'll create in a stacked format:

```
n = len(returns)
T = 21
tcols = list(range(T))
data = pd.DataFrame()
for i in range(n-T-1):
    df = returns.iloc[i:i+T+1]
    data = pd.concat([data, (df
                             .reset_index(drop=True)
                             .transpose()
                             .reset_index()
                             .assign(year=df.index[0].year,
                             month=df.index[0].month))],
                             ignore_index=True)
```

We'll separate a hold-out set with data for `2017`, the last full year with data:

```
train_data = data[:'2016']
test_data = data['2017']
```

For each train and test dataset, we generate a list with three input arrays containing the return series, the stock ticker (converted to integer values), and the month (as an integer), as shown here:

```
X_train = [
train_data.loc[:, list(range(1, window_size+1))].values.reshape(-1,
window_size , 1),
train_data.ticker,
train_data.filter(like='month')
]
y_train = train_data.label
[x.shape for x in X_train], y_train.shape
([(131917, 52, 1), (131917,), (131917, 12)], (131917,))
```

How to define the architecture

The functional API of Keras makes it easy to design architectures with multiple inputs and outputs. This example illustrates a network with three inputs, as follows:

- A two stacked LSTM layers with 25 and 10 units respectively
- An embedding layer that learns a 10-dimensional real-valued representation of the equities
- A one-hot encoded representation of the month

We begin by defining the three inputs with their respective shapes, as described here:

```
returns = Input(shape=(window_size, n_features), name='Returns')
tickers = Input(shape=(1,), name='Tickers')
months = Input(shape=(12,), name='Months')
```

To define stacked LSTM layers, we set the `return_sequences` keyword to `True`. This ensures that the first layer produces an output that conforms to the expected three-dimensional input format. Note that we also use dropout regularization and how the functional API passes the tensor outputs from one layer to the subsequent layer:

```
lstm1 = LSTM(units=lstm1_units,
             input_shape=(window_size, n_features),
             name='LSTM1',
             dropout=.2,
             return_sequences=True)(returns)
lstm_model = LSTM(units=lstm2_units,
                  dropout=.2,
                  name='LSTM2')(lstm1)
```

The embedding layer requires the `input_dim` keyword, which defines how many embeddings the layer will learn, the `output_dim` keyword, which defines the size of the embedding, and the `input_length` keyword to set the number of elements passed to the layer (here only one ticker per sample). To combine the embedding layer with the LSTM layer and the months input, we need to reshape (or flatten) it, as follows:

```
ticker_embedding = Embedding(input_dim=n_tickers,
output_dim=10,
input_length=1)(tickers)
ticker_embedding = Reshape(target_shape=(10,))(ticker_embedding)
```

Now we can concatenate the three tensors and add fully-connected layers to learn a mapping from these learned time series, ticker, and month indicators to the outcome, a positive or negative return in the following week, as shown here:

```
merged = concatenate([lstm_model, ticker_embedding, months],
                     name='Merged')
hidden_dense = Dense(10, name='FC1')(merged)
output = Dense(1, name='Output')(hidden_dense)
rnn = Model(inputs=[returns, tickers, months], outputs=output)
```

The summary lays out this slightly more sophisticated architecture with 29,371 parameters, as follows:

```
__________________________________________________________________________
Layer (type) Output Shape Param # Connected to
==========================================================================
Returns (InputLayer) (None, 52, 1) 0
__________________________________________________________________________
Tickers (InputLayer) (None, 1) 0
__________________________________________________________________________
LSTM1 (LSTM) (None, 52, 25) 2700 Returns[0][0]
__________________________________________________________________________
embedding_8 (Embedding) (None, 1, 10) 24890 Tickers[0][0]
__________________________________________________________________________
LSTM2 (LSTM) (None, 10) 1440 LSTM1[0][0]
__________________________________________________________________________
reshape_5 (Reshape) (None, 10) 0 embedding_8[0][0]
__________________________________________________________________________
Months (InputLayer) (None, 12) 0
__________________________________________________________________________
Merged (Concatenate) (None, 32) 0 LSTM2[0][0]
reshape_5[0][0]
Months[0][0]
__________________________________________________________________________
FC1 (Dense) (None, 10) 330 Merged[0][0]
__________________________________________________________________________
Output (Dense) (None, 1) 11 FC1[0][0]
==========================================================================
Total params: 29,371
Trainable params: 29,371
Non-trainable params: 0
```

We compile the model to compute a custom `auc` metric (see the notebook for implementation), as follows:

```
rnn.compile(loss='binary_crossentropy',
            optimizer='adam',
            metrics=['accuracy', auc])
```

We train the model for `50` epochs using early stopping, as shown here:

```
result = rnn.fit(X_train,
                 y_train,
                 epochs=50,
                 batch_size=32,
                 validation_data=(X_test, y_test),
                 callbacks=[checkpointer, early_stopping],
                 verbose=1
```

Training stops after 18 epochs, producing a test **area under the curve** (**AUC**) of 0.63 for the best model with 13 rounds of training (each of which takes around an hour on a single GPU).

Multivariate time-series regression

So far, we have limited our modeling efforts to single time series. RNNs are naturally well suited to multivariate time series and represent a non-linear alternative to the **Vector Autoregressive** (**VAR**) models we covered in `Chapter 8`, *Time Series Models*. See the `multivariate_timeseries` notebook for implementation details.

Loading the data

For comparison, we illustrate the application of RNNs to modeling and forecasting several time series using the same dataset we used for the VAR example, monthly data on consumer sentiment, and industrial production from the Federal Reserve's FRED service, as follows:

```
df = web.DataReader(['UMCSENT', 'IPGMFN'], 'fred', '1980', '2017-
                    12').dropna()
df.columns = ['sentiment', 'ip']
df.info()
DatetimeIndex: 456 entries, 1980-01-01 to 2017-12-01
Data columns (total 2 columns):
     sentiment 456 non-null float64
     ip 456 non-null float64
```

Preparing the data

We apply the same transformation—annual difference for both series, prior log-transform for industrial production—to achieve stationarity (see `Chapter 8`, *Time Series Models*, for details), as shown here:

```
df_transformed = pd.DataFrame({'ip': np.log(df.ip).diff(12),
                   'sentiment': df.sentiment.diff(12)}).dropna()
```

The `create_multivariate_rnn_data` function transforms a dataset of several time series into the shape required by the Keras RNN layers, namely `n_samples` x `window_size` x `n_series`, as follows:

```
defcreate_multivariate_rnn_data(data, window_size):
y = data[window_size:]
n = data.shape[0]
X = np.stack([data[i: j] for i, j in enumerate(range(window_size, n))],
             axis=0)
return X, y
```

We will use `window_size` of 24 months and obtain the desired inputs for our RNN model, as follows:

```
X, y = create_multivariate_rnn_data(df_transformed,
                                    window_size=window_size)
X.shape, y.shape
((420, 24, 2), (420, 2))
```

Finally, we split our data into a train and a test set, using the last 24 months to test the out-of-sample performance, as shown here:

```
test_size =24
train_size = X.shape[0]-test_size

X_train, y_train = X[:train_size], y[:train_size]
X_test, y_test = X[train_size:], y[train_size:]

X_train.shape, X_test.shape
((396, 24, 2), (24, 24, 2))
```

Defining and training the model

We use a similar architecture with two stacked LSTM layers with `12` and `6` units, respectively, followed by a fully-connected layer with 10 units. The output layer has two units, one for each time series. We compile them using mean absolute loss and the recommended `RMSProp` optimizer, as follows:

```
n_features = output_size = 2
lstm1_units = 12
lstm2_units = 6
rnn = Sequential([
LSTM(units=lstm1_units,
dropout=.2,
recurrent_dropout=.2,
input_shape=(window_size, n_features), name='LSTM1',
return_sequences=True),
LSTM(units=lstm2_units,
dropout=.2,
recurrent_dropout=.2,
name='LSTM2'),
Dense(10, name='FC1'),
Dense(output_size, name='Output')])

rnn.compile(loss='mae', optimizer='RMSProp')
```

The model has 1,268 parameters, as shown here:

```
rnn.summary()
_________________________________________________________
Layer (type) Output Shape Param #
=========================================================
LSTM1 (LSTM) (None, 24, 12) 720
_________________________________________________________
LSTM2 (LSTM) (None, 6) 456
_________________________________________________________
FC1 (Dense) (None, 10) 70
_________________________________________________________
Output (Dense) (None, 2) 22
=========================================================
Total params: 1,268
Trainable params: 1,268
Non-trainable params: 0
```

We train for `50` epochs with a `batch_size` value of `20` using early stopping:

```
result = rnn.fit(X_train,
                 y_train,
                 epochs=50,
                 batch_size=20,
                 validation_data=(X_test, y_test),
                 callbacks=[checkpointer, early_stopping],
                 verbose=1)
```

Training stops early after 25 epochs, yielding a test MAE of 1.71, which compares favorably to the test MAE for the VAR model of 1.91.

However, the two results are not fully comparable because the RNN model produces 24 one-step-ahead forecasts, whereas the VAR model uses its own predictions as input for its out-of-sample forecast. You may want to tweak the VAR setup to obtain comparable forecasts and compare their performance:

$$cov_{i,j} = \frac{\sum_{k=1}^{N}(x_{ik} - \bar{x_i})(x_{jk} - \bar{x_j})}{N-1}$$

The preceding formula highlights train and validation errors, and the out-of-sample predictions for both series.

LSTM and word embeddings for sentiment classification

RNNs are commonly applied to various natural language processing tasks. We've already encountered sentiment analysis using text data in part three of this book.

We are now going to illustrate how to apply an RNN model to text data to detect positive or negative sentiment (which can easily be extended to a finer-grained sentiment scale).

We are going to use word embeddings to represent the tokens in the documents. We covered word embeddings in `Chapter 15`, *Word Embeddings*. They are an excellent technique to convert text into a continuous vector representation such that the relative location of words in the latent space encodes useful semantic aspects based on the words' usage in context.

We saw in the previous RNN example that Keras has a built-in embedding layer that allows us to train vector representations specific to the task at hand. Alternatively, we can use pretrained vectors.

Loading the IMDB movie review data

To keep the data manageable, we will illustrate this use case with the IMDB reviews dataset, which contains 50,000 positive and negative movie reviews evenly split into a train and a test set, and with balanced labels in each dataset. The vocabulary consists of 88,586 tokens.

The dataset is bundled into Keras and can be loaded so that each review is represented as an integer-encoded sequence. We can limit the vocabulary to `num_words` while filtering out frequent and likely less informative words using `skip_top`, as well as sentences longer than `maxlen`. We can also choose `oov_char`, which represents tokens we chose to exclude from the vocabulary on frequency grounds, as follows:

```
from keras.datasets import imdb

vocab_size = 20000
(X_train, y_train), (X_test, y_test) = imdb.load_data(seed=42,
skip_top=0,
maxlen=None,
oov_char=2,
index_from=3
num_words=vocab_size)
```

In the second step, convert the lists of integers into fixed-size arrays that we can stack and provide as input to our RNN. The `pad_sequence` function produces arrays of equal length, truncated, and padded to conform to `maxlen`, as follows:

```
maxlen = 100

X_train_padded = pad_sequences(X_train,
truncating='pre',
padding='pre',
maxlen=maxlen)
```

Defining embedding and RNN architectures

Now we can set our RNN architecture. The first layer learns the word embeddings. We define the embedding dimension as previously using the `input_dim` keyword to set the number of tokens that we need to embed, the `output_dim` keyword, which defines the size of each embedding, and how long each input sequence is going to be.

Note that we are using GRUs this time, which train faster and perform better on smaller data. We are also using dropout for regularization, as follows:

```
embedding_size = 100
rnn = Sequential()
rnn.add(Embedding(input_dim=vocab_size, output_dim= embedding_size,
input_length=maxlen))
rnn.add(GRU(units=32, dropout=0.2, recurrent_dropout=0.2))
rnn.add(Dense(1, activation='sigmoid'))
```

The resulting model has over 2 million parameters, as shown here:

```
rnn.summary()
```

```
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
embedding_1 (Embedding) (None, 100, 100) 2000000
_________________________________________________________________
gru_1 (GRU) (None, 32) 12768
_________________________________________________________________
dense_1 (Dense) (None, 1) 33
=================================================================
Total params: 2,012,801
Trainable params: 2,012,801
Non-trainable params: 0
_________________________________________________________________
```

We compile the model to use our custom AUC metric, which we introduced previously, and train with early stopping:

```
rnn.fit(X_train_padded,
        y_train,
        batch_size=32,
        epochs=25,
        validation_data=(X_test_padded, y_test),
        callbacks=[checkpointer, early_stopping],
        verbose=1)
```

Training stops after eight epochs and we recover the weights for the best models to find a high test AUC of `0.9346`:

```
rnn.load_weights(rnn_path)
y_score = rnn.predict(X_test_padded)
roc_auc_score(y_score=y_score.squeeze(), y_true=y_test)
0.9346154079999999
```

Sentiment analysis with pretrained word vectors

In `Chapter 15`, *Word Embeddings*, we discussed how to learn domain-specific word embeddings. Word2vec, and related learning algorithms, produce high-quality word vectors, but require large datasets. Hence, it is common that research groups share word vectors trained on large datasets, similar to the weights for pretrained deep learning models that we encountered in the section on transfer learning in the previous chapter.

We are now going to illustrate how to use pretrained **Global Vectors for Word Representation** (**GloVe**) provided by the Stanford NLP group with the IMDB review dataset (see GitHub for references and the `sentiment_analysis_pretrained_embeddings` notebook for implementation details).

Preprocessing the text data

We are going to load the IMDB dataset from the source to manually preprocess (see notebook).

Keras provides a tokenizer that we use to convert the text documents to integer-encoded sequences, as shown here:

```
num_words = 10000
t = Tokenizer(num_words=num_words,
              lower=True,
              oov_token=2)
t.fit_on_texts(train_data.review)
vocab_size = len(t.word_index) + 1
train_data_encoded = t.texts_to_sequences(train_data.review)
test_data_encoded = t.texts_to_sequences(test_data.review)
```

We also use the `pad_sequences` function to convert the list of lists (of unequal length) to stacked sets of padded and truncated arrays for both the train and test datasets:

```
max_length = 100
X_train_padded = pad_sequences(train_data_encoded,
maxlen=max_length,
padding='post',
truncating='post')
y_train = train_data['label']
X_train_padded.shape
(25000, 100)
```

Loading the pretrained GloVe embeddings

We've downloaded and unzipped the GloVe data to the location indicated in the code and will now create a dictionary that maps GloVe tokens to 100-dimensional real-valued vectors, as follows:

```
glove_path = Path('data/glove/glove.6B.100d.txt')
embeddings_index = dict()

for line in glove_path.open(encoding='latin1'):
    values = line.split()
    word = values[0]
coefs = np.asarray(values[1:], dtype='float32')
embeddings_index[word] = coefs
```

There are around 340,000 word vectors that we use to create an embedding matrix that matches the vocabulary so that the RNN model can access embeddings by the token index:

```
embedding_matrix = np.zeros((vocab_size, 100))
for word, i in t.word_index.items():
embedding_vector = embeddings_index.get(word)
if embedding_vector isnotNone:
embedding_matrix[i] = embedding_vector
```

The difference between this and the RNN setup in the previous example is that we are going to pass the embedding matrix to the embedding layer and set it to non-trainable, so that the weights remain fixed during training:

```
rnn = Sequential([
Embedding(input_dim=vocab_size,
          output_dim= embedding_size,
          input_length=max_length,
          weights=[embedding_matrix],
          trainable=False),
GRU(units=32, dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')])
```

From here on we proceed as before. Training continues for 25 epochs and we obtain a test AUC score of `0.9106`. This is slightly worse than our result in the previous sections where we learned custom embedding for this domain, underscoring the value of training your own word embeddings.

You may want to apply these techniques to the larger financial text datasets that we used in part three.

Summary

In this chapter, we presented the specialized RNN architecture, which is tailored to sequential data. We covered how RNNs work, analyzed a computational graph, and saw how RNNs enable parameter sharing over numerous steps to capture long-range dependencies that FFNNs and CNNs are not well suited for.

We also reviewed the challenges of vanishing and exploding gradients and saw how gate units such as LSTM cells enable RNNs to learn dependencies over hundreds of time steps. Finally, we applied RNN models to challenges common in algorithmic trading, such as predicting univariate and multivariate time series and sentiment analysis.

In the next chapter, we will introduce unsupervised deep learning techniques, such as autoencoders and **Generative Adversarial Networks** (**GANs**), and their applications in investment and trading strategies.