19
RNNs for Multivariate Time Series and Sentiment Analysis

The previous chapter showed how **convolutional neural networks (CNNs)** are designed to learn features that represent the spatial structure of grid-like data, especially images, but also time series. This chapter introduces **recurrent neural networks (RNNs)** that specialize in sequential data where patterns evolve over time and learning typically requires memory of preceding data points.

Feedforward neural networks (FFNNs) treat the feature vectors for each sample as independent and identically distributed. Consequently, they do not take prior data points into account when evaluating the current observation. In other words, they have no memory.

The one- and two-dimensional convolutional filters used by CNNs can extract features that are a function of what is typically a small number of neighboring data points. However, they only allow shallow parameter-sharing: each output results from applying the same filter to the relevant time steps and features.

The major innovation of the RNN model is that each output is a function of both the previous output and new information. RNNs can thus incorporate information on prior observations into the computation they perform using the current feature vector. This recurrent formulation enables parameter-sharing across a much deeper computational graph (Goodfellow, Bengio, and Courville, 2016). In this chapter, you will 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.

Successful RNN use cases include various tasks that require mapping one or more input sequences to one or more output sequences and prominently feature natural language applications. We will explore how RNNs can be applied to univariate and multivariate time series to predict asset prices using market or fundamental data. We will also cover how RNNs can leverage alternative text data using word embeddings, which we covered in *Chapter 16*, *Word Embeddings for Earnings Calls and SEC Filings*, to classify the sentiment expressed in documents. Finally, we will use the most informative sections of SEC filings to learn word embeddings and predict returns around filing dates.

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

- How recurrent connections allow RNNs to memorize patterns and model a hidden state
- Unrolling and analyzing the computational graph of RNNs
- How gated units learn to regulate RNN memory from data to enable long-range dependencies
- Designing and training RNNs for univariate and multivariate time series in Python
- How to learn word embeddings or use pretrained word vectors for sentiment analysis with RNNs
- Building a bidirectional RNN to predict stock returns using custom word embeddings

You can find the code examples and additional resources in the GitHub repository's directory for this chapter.

How recurrent neural nets work

RNNs assume that the input data has been generated as a sequence such that previous data points impact the current observation and are relevant for predicting subsequent values. Thus, they allow more complex input-output relationships than FFNNs and CNNs, which are designed to map one input vector to one output vector 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. For a

good overview, refer to *Chapter 10* in Goodfellow, Bengio, and Courville (2016).

The diagram in *Figure 19.1*, inspired by Andrew Karpathy's 2015 blog post *The Unreasonable Effectiveness of Recurrent Neural Networks* (see GitHub for a link), illustrates mappings from input to output vectors using non-linear transformations carried out by one or more neural network layers:

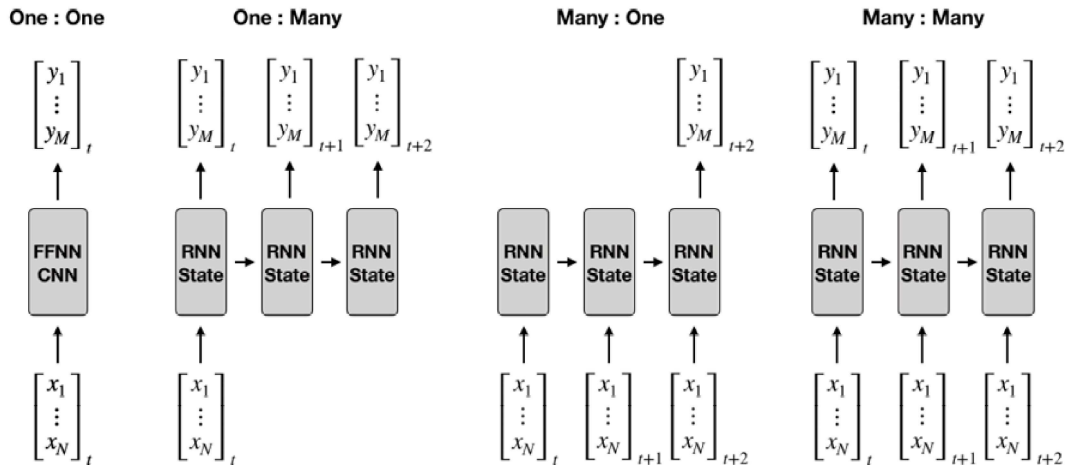


Figure 19.1: Various types of sequence-to-sequence models

The left panel shows a one-to-one mapping between vectors of fixed sizes, typical for FFNs and CNNs covered in the last two chapters. The other three panels show various RNN applications that map input vectors to output vectors by applying a recurrent transformation to the new input and the state produced by the previous iteration. The *x* input vectors to an RNN are also called **context**.

The vectors are time-indexed, as usually required by 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 include:

- **One-to-many**: Image captioning, for example, takes a single vector of pixels (as in the previous chapter) and maps it to a sequence of words.
- **Many-to-one**: Sentiment analysis takes a sequence of words or tokens (see *Chapter 14*, *Text Data for Trading – Sentiment Analysis*) and maps it to an output scalar or vector.

- **Many-to-many**: Machine translation or labeling of video frame map sequences of input vectors to sequences of output vectors, either in a synchronized (as shown) or asynchronous fashion. Multistep 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 that is fixed but learned from the data can be applied as many times as needed.

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

Unfolding a computational graph with cycles

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

Figure 19.2 shows the **computational graph** implied by a single hidden RNN unit that learns two weight matrices during training:

- W_{hh}: applied to the previous hidden state, h_{t-1}
- W_{hx}: applied to the current input, x_t

The RNN's output, y_t, is a nonlinear transformation of the sum of the two matrix multiplications using, for example, the tanh or ReLU activation functions:

$$y_t = g(W_{hh}h_{t-1} + W_{xh}x_t)$$

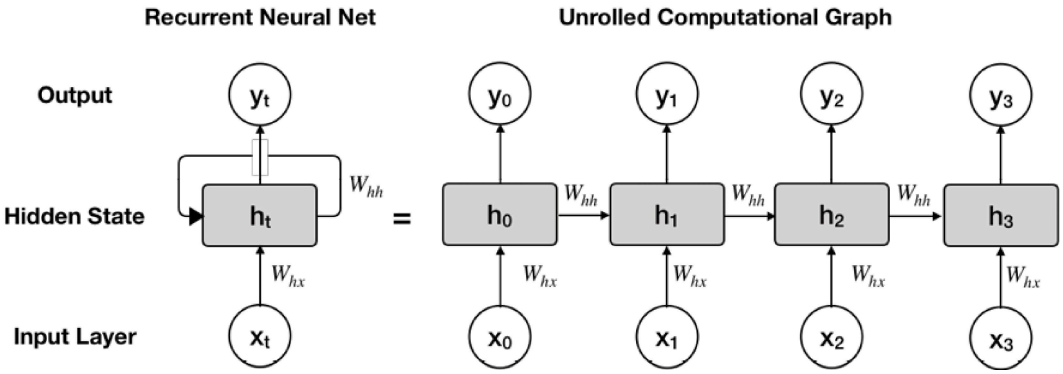


Figure 19.2: Recurrent and unrolled view of the computational graph of an RNN with a single hidden unit

The right side of the equation shows the effect of unrolling the recurrent relationship depicted in the right panel of the figure. It highlights the repeated linear algebra transformations and the resulting hidden 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 this baseline architecture in the subsequent section.

Backpropagation through time

The unrolled computational graph in the preceding figure highlights that the learning process necessarily encompasses all time steps of the given input sequence. The backpropagation algorithm that updates the weights during training involves a forward pass from left to right along with the unrolled computational graph, followed by a backward pass in the opposite direction.

As discussed in *Chapter 17*, *Deep Learning for Trading*, the backpropagation algorithm evaluates a loss function and computes its gradient with respect to the parameters to update the weights accordingly. 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. Therefore, the algorithm is called **backpropagation through time** (Werbos 1990).

It highlights both the power of an RNN to model long-range dependencies by sharing parameters across an arbitrary number of sequence elements while 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 FFNN and CNN architectures we covered in the previous two chapters, RNNs can be optimized in a variety of ways to capture the dynamic relationship between input and output data.

In addition to modifying the recurrent connections between the hidden states, alternative approaches include recurrent output relationships, bidirectional RNNs, and encoder-decoder architectures. Refer to GitHub for background references to complement this brief summary.

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 are now decoupled and can be trained in parallel.

However, to successfully learn relevant past information, the training output samples need to reflect this information so that backpropagation can adjust the network parameters accordingly. To the extent that asset returns are independent of their lagged values, financial data may not meet this requirement. The use of previous outcome values alongside the input vectors is called **teacher forcing** (Williams and Zipser, 1989).

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

Bidirectional RNNs

For some tasks, it can be realistic and beneficial for the output to depend not only on past sequence elements, but also on future elements (Schuster and Paliwal, 1997). Machine translation or speech and handwriting recognition are examples where subsequent sequence elements are both informative and realistically available to disambiguate competing outputs.

For a one-dimensional sequence, **bidirectional RNNs** combine an RNN 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 in the natural language and music domains (Sigtia et al., 2014) have been very successful (see *Chapter 16*, *Word Embeddings for Earnings Calls and SEC Filings*, and the last example in this chapter using SEC filings).

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, attention, and transformers

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 other applications with this characteristic (Prabhavalkar et al., 2017).

The **encoder** is an RNN that maps the input space to a different space, also called **latent space**, whereas the **decoder** function is a complementary RNN that maps the encoded input to the target space (Cho et al., 2014). In the next chapter, we will cover autoencoders that learn a feature representation in an unsupervised setting using a variety of deep learning architectures.

Encoder and decoder RNNs are trained jointly so that the input of the final encoder hidden state becomes the input to the decoder, 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 16*, *Word Embeddings for Earnings Calls and SEC Filings*), stores the result, and uses a weighted average of these feature vectors as context. The weights are learned by the model and alternate between putting more weight or attention to different elements of the input.

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 (Vaswani et al., 2017).

How to design deep RNNs

The **unrolled computational graph** in *Figure 19.2* shows that each transformation involves a linear matrix operation followed by a nonlinear transformation that could be jointly represented by a single network layer.

In the two preceding chapters, we saw how adding depth allows FFNNs, and CNNs in particular, to learn more useful hierarchical representations. RNNs also benefit from decomposing the input-output mapping into multiple layers. For RNNs, this mapping typically transforms:

- The input and the prior hidden state into the current hidden state
- The hidden state into the output

A common approach is to **stack recurrent layers** on top of each other so that they 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. We will demonstrate this approach in the next section.

Less popular alternatives include adding layers to the connections from input to the hidden state, between hidden states, or from the hidden state

to the output. These designs employ skip connections to avoid a situation where the shortest path between time steps increases and training becomes 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 far apart from the current observation (Hochreiter et al., 2001).

The fundamental problem is the impact of repeated multiplication on gradients during backpropagation over many time steps. As a result, the **gradients tend to either vanish** and decrease toward zero (the typical case), **or explode** and grow toward infinity (less frequent, but rendering optimization very difficult).

Even if parameters allow stability and the network is able to store memories, long-term interactions will receive exponentially smaller weights due to the multiplication of many Jacobians, the matrices containing the gradient information. Experiments have shown that stochastic gradient descent faces serious challenges in training RNNs for sequences with only 10 or 20 elements.

Several RNN design techniques have been introduced to address this challenge, including **echo state networks** (Jaeger, 2001) and **leaky units** (Hihi and Bengio, 1996). The latter operate 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. Other strategies include connections that skip time steps or units that integrate signals from different frequencies.

The most successful approaches use gated units that are trained to regulate how much past information a unit maintains in its current state and when to reset or forget this information. As a result, they are able to learn

dependencies over hundreds of time steps. The most popular examples include **long short-term memory (LSTM)** units and **gated recurrent units (GRUs)**. An empirical comparison by Chung et al. (2014) finds both units superior to simpler recurrent units such as tanh units, while performing equally well on various speech and music modeling tasks.

Long short-term memory – learning how much to forget

RNNs with an LSTM architecture have more complex units that maintain an internal state. They contain gates to keep track of dependencies between elements of the input sequence and regulate the cell's state accordingly. These gates recurrently connect to each other instead of the hidden units we encountered earlier. They aim to address the problem of vanishing and exploding gradients due to the repeated multiplication of possibly very small or very large values by letting gradients pass through unchanged (Hochreiter and Schmidhuber, 1996).

The diagram in *Figure 19.3* shows the information flow for an unrolled LSTM unit and outlines its typical gating mechanism:

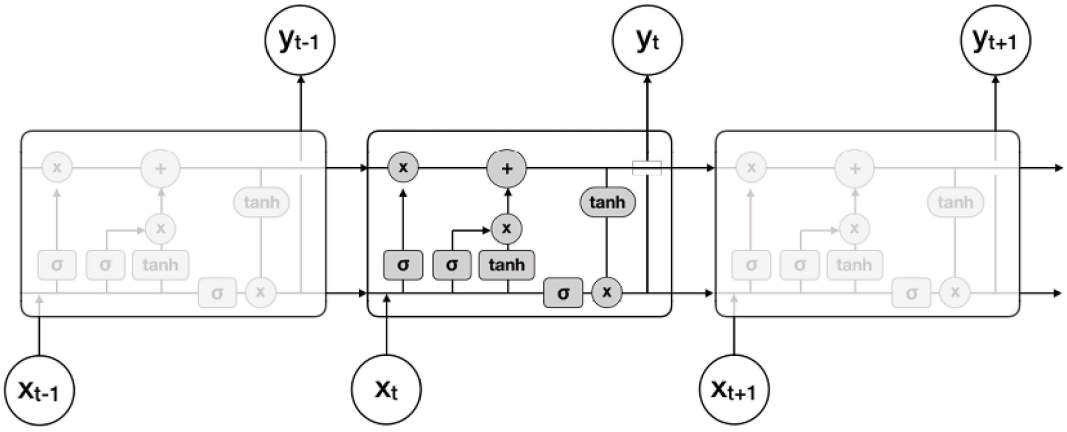
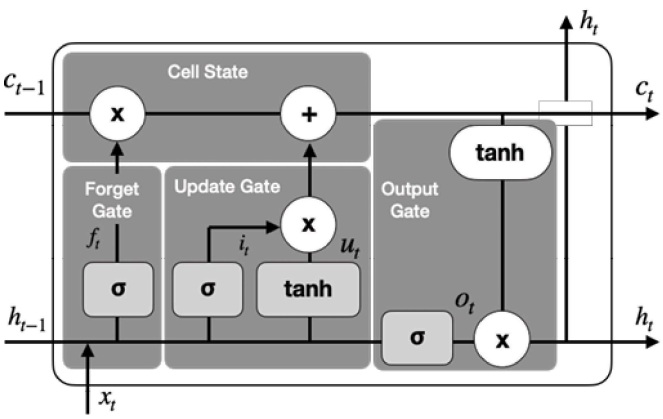


Figure 19.3: Information flow through an unrolled LSTM cell

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 white nodes in *Figure 19.4* identify ele-

ment-wise operations, and the gray elements represent layers with weight and bias parameters learned during training:



LSTM Equations

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

$$u_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot u_t$$

$$h_t = o_t \odot \tanh c_t$$

Figure 19.4: The logic of, and math behind, an LSTM cell

The **cell state**, *c*, passes along 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:

1. The **forget gate** controls how much of the cell's state should be voided to regulate the network's memory. 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, which has been normalized to the [0, 1] range, by the cell state, reducing or keeping it accordingly.
2. The **input gate** also computes a sigmoid activation from h_{t-1} and x_t that produces update candidates. A tan_h activation in the range from [-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 tan_h activation.

Gated recurrent units

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

GRUs aim for each recurrent unit to adaptively capture dependencies of different time scales. Similar to the LSTM unit, the GRU has gating units that modulate the flow of information inside the unit but discard separate memory cells (see references on GitHub for additional details).

RNNs for time series with TensorFlow 2

In this section, we illustrate how to build recurrent neural nets using the TensorFlow 2 library for various scenarios. The first set of models includes the 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 16, Word Embeddings for Earnings Calls and SEC Filings*).

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'll build a **deep RNN** with three distinct inputs to classify asset price movements. To this end, we'll combine a two-layer, **stacked LSTM** with learned **embeddings** and one-hot encoded categorical data. Finally, we will demonstrate how to model **multivariate time series** using an RNN.

Univariate regression – predicting the S&P 500

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

We'll obtain data for 2010-2019 from the Federal Reserve Bank's Data Service (FRED; see *Chapter 2, Market and Fundamental Data – Sources and Techniques*):

```
sp500 = web.DataReader('SP500', 'fred', start='2010', end='2020').dropna()
sp500.info()
DatetimeIndex: 2463 entries, 2010-03-22 to 2019-12-31
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   SP500   2463 non-null   float64
```

We preprocess the data by scaling it to the [0, 1] interval using scikit-learn's `MinMaxScaler()` class:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
sp500_scaled = pd.Series(scaler.fit_transform(sp500).squeeze(),
                         index=sp500.index)
```

How to get time series data into shape for an RNN

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

The input to every LSTM layer must have three dimensions, namely:

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

The following figure visualizes the shape of the input tensor:

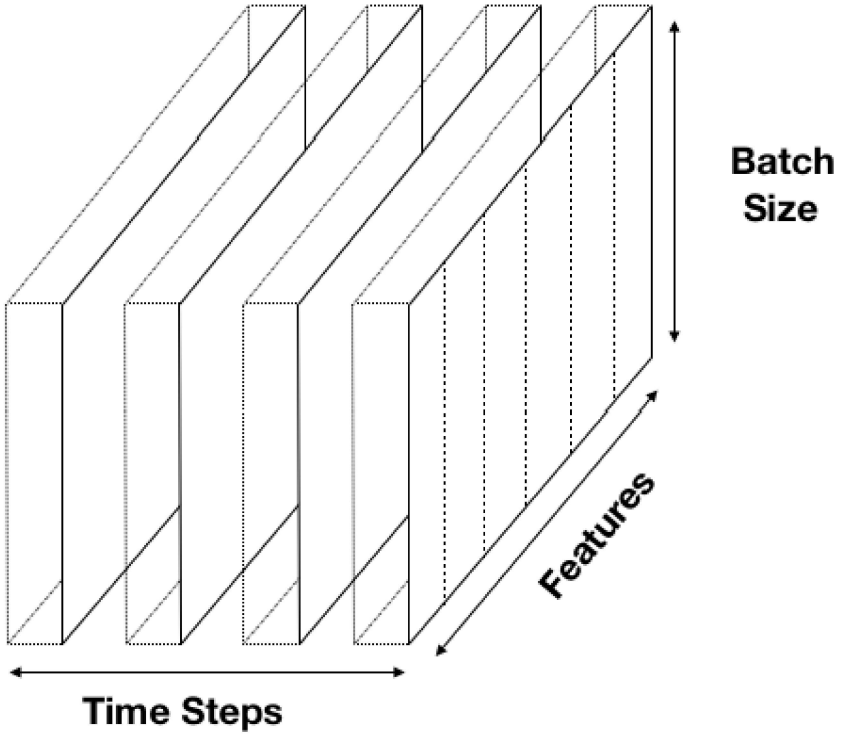


Figure 19.5: The three dimensions of an RNN input tensor

Our S&P 500 sample has 2,463 observations or time steps. We will create overlapping sequences using a window of 63 observations each. Using a simpler window of size $T = 5$ to illustrate this autoregressive sequence pattern, we obtain input-output pairs where each output is associated with its first five lags, as shown in the following table:

Input	Output
$\langle x_1, x_2, x_3, x_4, x_5 \rangle$	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

Figure 19.6: Input-output pairs with a T=5 size window

We can use the `create_univariate_rnn_data()` function to stack the overlapping sequences that we select using a rolling window:

```
def create_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 using `window_size=63` to obtain a two-dimensional dataset with a shape of the number of samples x the number of time steps:

```
X, y = create_univariate_rnn_data(sp500_scaled, window_size=63)
X.shape
(2356, 63)
```

We will use data from 2019 as our test set and reshape the features to add a requisite third dimension:

```
X_train = X[:'2018'].values.reshape(-1, window_size, 1)
y_train = y[:'2018']
# keep the last year for testing
X_test = X['2019'].values.reshape(-1, window_size, 1)
y_test = y['2019']
```

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

Now that we have created autoregressive input/output pairs from our time series and split the pairs into training and test sets, we can define our RNN architecture. The Keras interface of TensorFlow 2 makes it very straightforward to build an RNN with two hidden layers with the following specifications:

- **Layer 1**: An LSTM module with 10 hidden units (with `input_shape = (window_size,1)` ; we will define `batch_size` in the omitted first dimension during training)

- **Layer 2**: A fully connected module with a single unit and linear activation
- **Loss**: `mean_squared_error` to match the regression objective

Just a few lines of code create the computational graph:

```
rnn = Sequential([
    LSTM(units=10,
         input_shape=(window_size, n_features), name='LSTM'),
    Dense(1, name='Output')
])
```

The summary shows that the model has 491 parameters:

```
rnn.summary()
Layer (type)                 Output Shape              Param #
LSTM (LSTM)                  (None, 10)                480
Output (Dense)               (None, 1)                 11
Total params: 491
Trainable params: 491
```

Training and evaluating the model

We train the model using the RMSProp optimizer recommended for RNN with default settings and compile the model with `mean_squared_error` for this regression problem:

```
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 and train the model for 500 episodes:

```
early_stopping = EarlyStopping(monitor='val_loss',
                               patience=50,
                               restore_best_weights=True)
lstm_training = 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 138 epochs. The loss history in *Figure 19.7* shows the 5-epoch rolling average of the training and validation RMSE, highlights the best epoch, and shows that the loss is 0.998 percent:

```
loss_history = pd.DataFrame(lstm_training.history).pow(.5)
loss_history.index += 1
best_rmse = loss_history.val_loss.min()
best_epoch = loss_history.val_loss.idxmin()
loss_history.columns=['Training RMSE', 'Validation RMSE']
title = f'Best Validation RMSE: {best_rmse:.4%}'
loss_history.rolling(5).mean().plot(logy=True, lw=2, title=title, ax=ax)
```

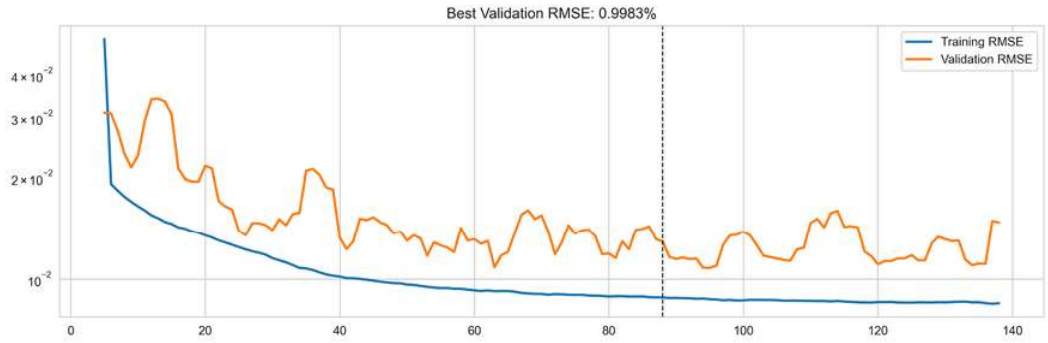


Figure 19.7: Cross-validation performance

Re-scaling the predictions

We use the `inverse_transform()` method of `MinMaxScaler()` to rescale the model predictions to the original S&P 500 range of values:

```
test_predict_scaled = rnn.predict(X_test)
test_predict = (pd.Series(scaler.inverse_transform(test_predict_scaled)
                          .squeeze(),
                          index=y_test.index))
```

The four plots in *Figure 19.8* illustrate the forecast performance based on the rescaled predictions that track the 2019 out-of-sample S&P 500 data with a test **information coefficient** (**IC**) of 0.9889:

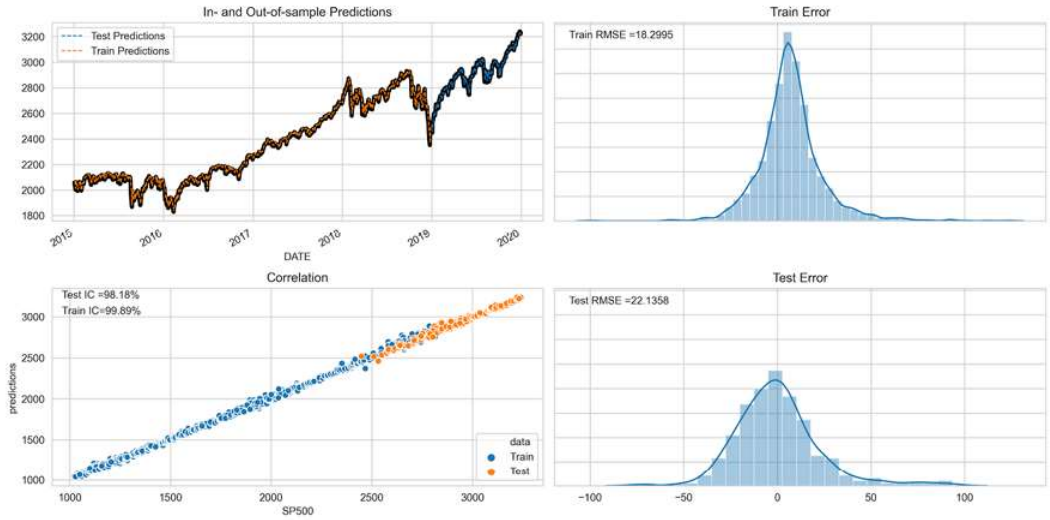


Figure 19.8: RNN performance on S&P 500 predictions

Stacked LSTM – predicting price moves and returns

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

Figure 19.9 outlines the architecture that illustrates how to combine different data sources in a single deep neural network. For example, instead of, or in addition to, one-hot encoded months, you could add technical or fundamental features:

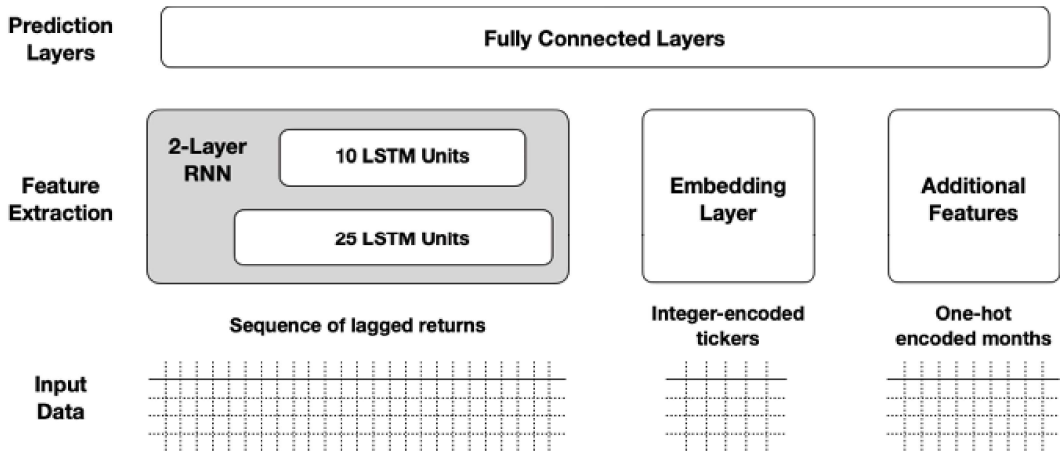


Figure 19.9: Stacked LSTM architecture with additional features

Preparing the data – how to create weekly stock returns

We load the Quandl adjusted stock price data (see instructions on GitHub on how to obtain the source data) as follows (refer to the `build_dataset.ipynb` 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 with complete data for the 2008-17 period:

```
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 create and stack rolling sequences of 52 weekly returns for each ticker and week as follows:

```
n = len(returns)
T = 52
tcols = list(range(T))
tickers = returns.columns
data = pd.DataFrame()
for i in range(n-T-1):
    df = returns.iloc[i:i+T+1]
    date = df.index.max()
    data = pd.concat([data, (df.reset_index(drop=True).T
                             .assign(date=date, ticker=tickers)
                             .set_index(['ticker', 'date']))])
```

We winsorize outliers at the 1 and 99 percentile level and create a binary label that indicates whether the weekly return was positive:

```
data[tcols] = (data[tcols].apply(lambda x: x.clip(lower=x.quantile(.01),
                                                  upper=x.quantile(.99))))
data['label'] = (data['fwd_returns'] > 0).astype(int)
```

As a result, we obtain 1.16 million observations on over 2,400 stocks with 52 weeks of lagged returns each (plus the label):

```
data.shape
(1167341, 53)
```

Now we are ready to create the additional features, split the data into training and test sets, and bring them into the three-dimensional format required for the LSTM.

How to create multiple inputs in RNN format

This example illustrates how to combine several input data sources, namely:

- Rolling sequences of 52 weeks of lagged returns
- One-hot encoded indicator variables for each of the 12 months
- Integer-encoded values for the tickers

The following code generates the two additional features:

```
data['month'] = data.index.get_level_values('date').month
data = pd.get_dummies(data, columns=['month'], prefix='month')
data['ticker'] = pd.factorize(data.index.get_level_values('ticker'))[0]
```

Next, we create a training set covering the 2009-2016 period and a separate test set with data for 2017, the last full year with data:

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

For training and test datasets, we generate a list containing the three input arrays as shown in *Figure 19.9*:

- The lagged return series (using the format described in *Figure 19.5*)
- The integer-encoded stock ticker as a one-dimensional array
- The month dummies as a two-dimensional array with one column per month

```
window_size=52
sequence = list(range(1, window_size+1))
X_train = [
    train_data.loc[:, sequence].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
[(1035424, 52, 1), (1035424,), (1035424, 12)], (1035424,)
```

How to define the architecture using Keras' Functional API

Keras' Functional API makes it easy to design an architecture like the one outlined at the beginning of this section with multiple inputs (or several outputs, as in the SVHN example in *Chapter 18, CNNs for Financial Time Series and Satellite Images*). This example illustrates a network with three inputs:

1. **Two stacked LSTM layers** with 25 and 10 units, respectively
2. An **embedding layer** that learns a 10-dimensional real-valued representation of the equities
3. A **one-hot encoded** representation of the month

We begin by defining the three inputs with their respective shapes:

```
n_features = 1
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 for the first layer to `True`. This ensures that the first layer produces an output in 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's input:

```
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 TensorFlow 2 guide for RNNs highlights the fact that GPU support is only available when using the default values for most LSTM settings (https://www.tensorflow.org/guide/keras/rnn).

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
- The `input_length` parameter, which sets the number of elements passed to the layer (here, only one ticker per sample)

The goal of the embedding layer is to learn vector representations that capture the relative locations of the feature values to one another with respect to the outcome. We'll choose a five-dimensional embedding for the roughly 2,500 ticker values to combine the embedding layer with the LSTM layer and the month dummies we need to reshape (or flatten) it:

```
ticker_embedding = Embedding(input_dim=n_tickers,
                             output_dim=5,
                             input_length=1)(tickers)
ticker_embedding = Reshape(target_shape=(5,))(ticker_embedding)
```

Now we can concatenate the three tensors, followed by `BatchNormalization`:

```
merged = concatenate([lstm_model, ticker_embedding, months], name='Merged')
bn = BatchNormalization()(merged)
```

The fully connected final layers learn a mapping from these stacked LSTM layers, ticker embeddings, and month indicators to the binary outcome that reflects a positive or negative return over the following week. We formulate the complete RNN by defining its inputs and outputs with the implicit data flow we just defined:

```
hidden_dense = Dense(10, name='FC1')(bn)
output = Dense(1, name='Output', activation='sigmoid')(hidden_dense)
rnn = Model(inputs=[returns, tickers, months], outputs=output)
```

The summary lays out this slightly more sophisticated architecture with 16,984 parameters:

```
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 (Embedding)           (None, 1, 5)         12445       Tickers[0][0]
LSTM2 (LSTM)                    (None, 10)           1440        LSTM1[0][0]
reshape (Reshape)               (None, 5)           0          embedding[0][0]
Months (InputLayer)             [(None, 12)]         0
Merged (Concatenate)            (None, 27)           0           LSTM2[0][0]
                                                                 reshape[0][0]
                                                                 Months[0][0]
batch_normalization (BatchNorma (None, 27)           108         Merged[0][0]
FC1 (Dense)                     (None, 10)           280
atch_normalization[0][0]
Output (Dense)                  (None, 1)            11          FC1[0][0]
Total params: 16,984
Trainable params: 16,930
Non-trainable params: 54
```

We compile the model using the recommended RMSProp optimizer with default settings and compute the AUC metric that we'll use for early stopping:

```
optimizer = tf.keras.optimizers.RMSprop(lr=0.001,
                                        rho=0.9,
                                        epsilon=1e-08,
                                        decay=0.0)
rnn.compile(loss='binary_crossentropy',
            optimizer=optimizer,
            metrics=['accuracy',
                     tf.keras.metrics.AUC(name='AUC')])
```

We train the model for 50 epochs by using early stopping:

```
result = rnn.fit(X_train,
                 y_train,
```

```
                epochs=50,
                batch_size=32,
                validation_data=(X_test, y_test),
                callbacks=[early_stopping])
```

The following plots show that training stops after 8 epochs, each of which takes around three minutes on a single GPU. It results in a test AUC of 0.6816 and a test accuracy of 0.6193 for the best model:

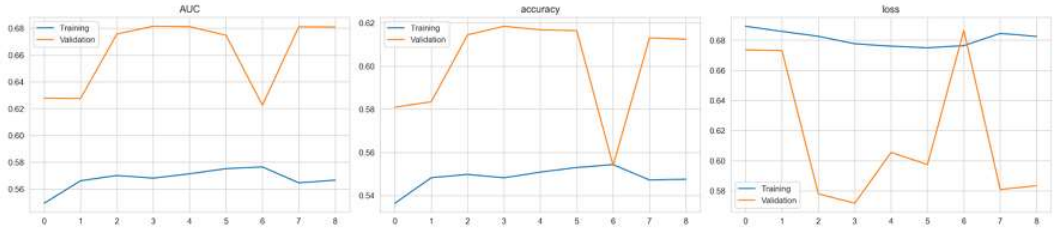


Figure 19.10: Stacked LSTM classification—cross-validation performance

The IC for the test prediction and actual weekly returns is 0.32.

Predicting returns instead of directional price moves

The `stacked_lstm_with_feature_embeddings_regression.ipynb` notebook illustrates how to adapt the model to the regression task of predicting returns rather than binary price changes.

The required changes are minor; just do the following:

1. Select the `fwd_returns` outcome instead of the binary `label`.
2. Convert the model output to linear (the default) instead of `sigmoid`.
3. Update the loss to mean squared error (and early stopping references).
4. Remove or update optional metrics to match the regression task.

Using otherwise the same training parameters (except that the Adam optimizer with default settings yields a better result in this case), the validation loss improves for nine epochs. The average weekly IC is 3.32, and 6.68 for the entire period while significant at the 1 percent level. The av-

erage weekly return differential between the equities in the top and bottom quintiles of predicted returns is slightly above 20 basis points:

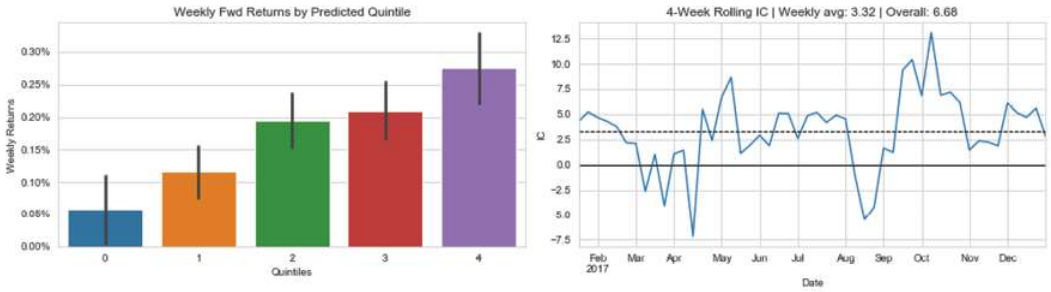


Figure 19.11: Stacked LSTM regression—out-of-sample performance

Multivariate time-series regression for macro data

So far, we have limited our modeling efforts to a single time series. RNNs are well-suited to multivariate time series and represent a nonlinear alternative to the **vector autoregressive (VAR)** models we covered in *Chapter 9, Time-Series Models for Volatility Forecasts and Statistical Arbitrage*. Refer to the `multivariate_timeseries` notebook for implementation details.

Loading sentiment and industrial production data

We'll show how to model and forecast multiple time series using RNNs with the same dataset we used for the VAR example. It has monthly observations over 40 years on consumer sentiment and industrial production from the Federal Reserve's FRED service:

```
df = web.DataReader(['UMCSENT', 'IPGMFN'], 'fred', '1980', '2019-12').dropna()
df.columns = ['sentiment', 'ip']
df.info()
DatetimeIndex: 480 entries, 1980-01-01 to 2019-12-01
Data columns (total 2 columns):
sentiment    480 non-null float64
ip           480 non-null float64
```

Making the data stationary and adjusting the scale

We apply the same transformation—annual difference for both series, prior log-transform for industrial production—to achieve stationarity (see *Chapter 9, Time-Series Models for Volatility Forecasts and Statistical Arbitrage* for details). We also rescale it to the [0, 1] range to ensure that the network gives both series equal weight during training:

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

Figure 19.12 displays the original and transformed macro time series:

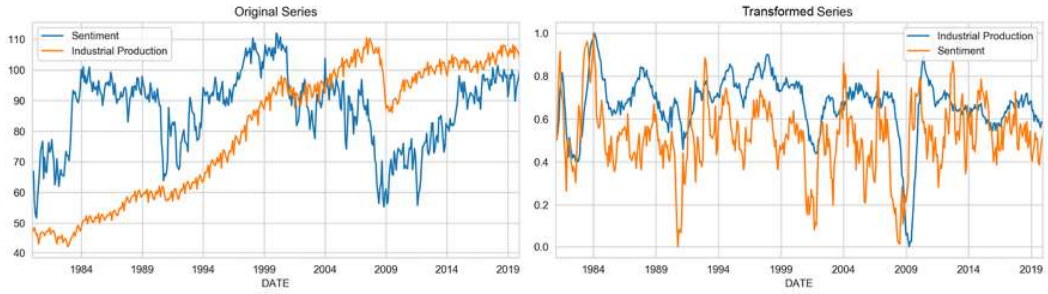


Figure 19.12: Original and transformed time series

Creating multivariate RNN inputs

The `create_multivariate_rnn_data()` function transforms a dataset of several time series into the three-dimensional shape required by TensorFlow's RNN layers, formed as `n_samples × window_size × n_series`:

```
def create_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
```

A `window_size` value of 18 ensures that the entries in the second dimension are the lagged 18 months of the respective output variable. We thus obtain the RNN model inputs for each of the two features as follows:

```
X, y = create_multivariate_rnn_data(df_transformed, window_size=window_size)
X.shape, y.shape
((450, 18, 2), (450, 2))
```

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

```
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
((426, 18, 2), (24, 18, 2))
```

Defining and training the model

Given the relatively small dataset, we use a simpler RNN architecture than in the previous example. It has a single LSTM layer with 12 units, followed by a fully connected layer with 6 units. The output layer has two units, one for each time series.

We compile using mean absolute loss and the recommended RMSProp optimizer:

```
n_features = output_size = 2
lstm_units = 12
dense_units = 6
rnn = Sequential([
    LSTM(units=lstm_units,
         dropout=.1,
         recurrent_dropout=.1,
         input_shape=(window_size, n_features), name='LSTM',
         return_sequences=False),
    Dense(dense_units, name='FC'),
```

```
    Dense(output_size, name='Output')
])
rnn.compile(loss='mae', optimizer='RMSProp')
```

The model still has 812 parameters, compared to 10 for the `VAR(1, 1)` model from *Chapter 9, Time-Series Models for Volatility Forecasts and Statistical Arbitrage*:

```
Layer (type)                 Output Shape              Param #
LSTM (LSTM)                  (None, 12)                720
FC (Dense)                   (None, 6)                 78
Output (Dense)               (None, 2)                 14
Total params: 812
Trainable params: 812
```

We train for 100 epochs with a `batch_size` of 20 using early stopping:

```
result = rnn.fit(X_train,
                 y_train,
                 epochs=100,
                 batch_size=20,
                 shuffle=False,
                 validation_data=(X_test, y_test),
                 callbacks=[checkpointer, early_stopping],
                 verbose=1)
```

Training stops early after 62 epochs, yielding a test MAE of 0.034, an almost 25 percent improvement over the test MAE for the VAR model of 0.043 on the same task.

However, the two results are not fully comparable because the RNN produces 18 1-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 the performance.

Figure 19.13 highlights training and validation errors, and the out-of-sample predictions for both series:

Figure 19.13: Cross-validation and test results for RNNs with multiple macro series

RNNs for text data

RNNs are commonly applied to various natural language processing tasks, from machine translation to sentiment analysis, that we already encountered in Part 3 of this book. In this section, we will illustrate how to apply an RNN to text data to detect positive or negative sentiment (easily extensible to a finer-grained sentiment scale) and to predict stock returns.

More specifically, we'll use word embeddings to represent the tokens in the documents. We covered word embeddings in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*. They are an excellent technique for converting a token into a dense, real-value vector because the relative location of words in the embedding space encodes useful semantic aspects of how they are used in the training documents.

We saw in the previous stacked RNN example that TensorFlow 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. We'll demonstrate both approaches in the following three sections.

LSTM with embeddings for sentiment classification

This example shows how to learn custom embedding vectors while training an RNN on the classification task. This differs from the word2vec model that learns vectors while optimizing predictions of neighboring tokens, resulting in their ability to capture certain semantic relationships

among words (see *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*). Learning word vectors with the goal of predicting sentiment implies that embeddings will reflect how a token relates to the outcomes it is associated with.

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 training set and a test set, with balanced labels in each dataset. The vocabulary consists of 88,586 tokens. Alternatively, you could use the much larger Yelp review data (after converting the text into numerical sequences; see the next section on using pretrained embeddings or TensorFlow 2 docs).

The dataset is bundled into TensorFlow 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 the `oov_char` value, which represents tokens we chose to exclude from the vocabulary on frequency grounds:

```
from tensorflow.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 an input to our RNN. The `pad_sequence` function produces arrays of equal length, truncated and padded to conform to `maxlen`:

```
maxlen = 100
X_train_padded = pad_sequences(X_train,
```

```
                                truncating='pre',
                                padding='pre',
                                maxlen=maxlen)
```

Defining embedding and the RNN architecture

Now we can set up our RNN architecture. The first layer learns the word embeddings. We define the embedding dimensions as before, using the following:

- The `input_dim` keyword, which sets the number of tokens that we need to embed
- The `output_dim` keyword, which defines the size of each embedding
- The `input_len` parameter, which specifies how long each input sequence is going to be

Note that we are using GRU units this time that train faster and perform better on smaller amounts of data. We are also using recurrent dropout for regularization:

```
embedding_size = 100
rnn = Sequential([
    Embedding(input_dim=vocab_size,
              output_dim= embedding_size,
              input_length=maxlen),
    GRU(units=32,
        dropout=0.2, # comment out to use optimized GPU implementation
        recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])
```

The resulting model has over 2 million trainable parameters:

```
Layer (type)                 Output Shape              Param #
embedding (Embedding)        (None, 100, 100)          2000000
gru (GRU)                    (None, 32)                12864
dense (Dense)                (None, 1)                 33
```

```
Total params: 2,012,897
Trainable params: 2,012,897
```

We compile the model to use the AUC metric 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=[early_stopping],
        verbose=1)
```

Training stops after 12 epochs, and we recover the weights for the best models to find a high test AUC of 0.9393:

```
y_score = rnn.predict(X_test_padded)
roc_auc_score(y_score=y_score.squeeze(), y_true=y_test)
0.9393289376
```

Figure 19.14 displays the cross-validation performance in terms of accuracy and AUC:

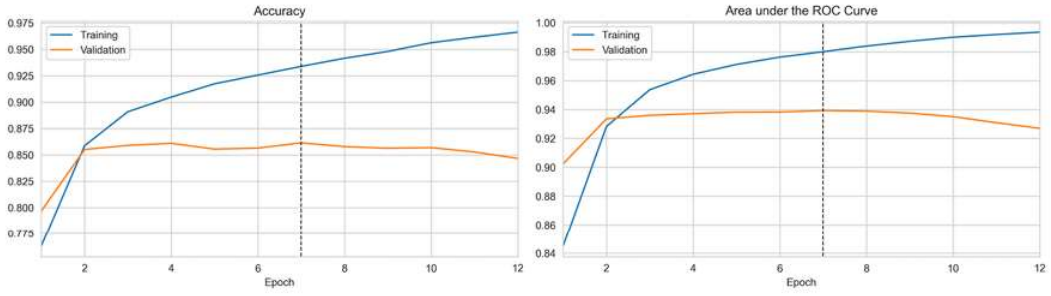


Figure 19.14: Cross-validation for RNN using IMDB data with custom embeddings

Sentiment analysis with pretrained word vectors

In *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, 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 (refer to 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 it (see the notebook). TensorFlow provides a `Tokenizer`, which we'll use to convert the text documents to integer-encoded sequences:

```
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 training and test data:

```
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 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:

```
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 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 is not None:
        embedding_matrix[i] = embedding_vector
```

Defining the architecture with frozen weights

The difference with 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 *not 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 32 epochs, as shown in *Figure 19.15*, 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:

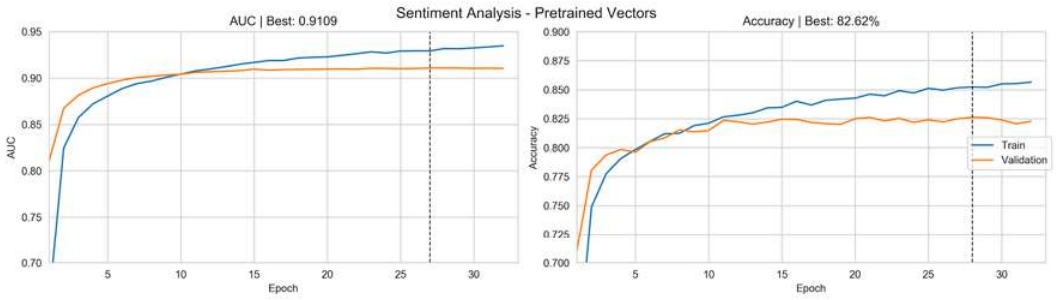


Figure 19.15: Cross-validation and test results for RNNs with multiple macro series

You may want to apply these techniques to the larger financial text datasets that we used in Part 3.

Predicting returns from SEC filing embeddings

In *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, we discussed important differences between product reviews and financial text data. While the former was useful to illustrate important workflows, in this section, we will tackle more challenging but also more relevant financial documents. More specifically, we will use the SEC filings data introduced in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, to learn word embeddings tailored to predicting the return of the ticker associated with the disclosures from before publication to one week after.

The `sec_filings_return_prediction` notebook contains the code examples for this section. See the `sec_preprocessing` notebook in *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*, and instructions in the data folder on GitHub on how to obtain the data.

Source stock price data using yfinance

There are 22,631 filings for the period 2013-16. We use yfinance to obtain stock price data for the related 6,630 tickers because it achieves higher coverage than Quandl's WIKI Data. We use the ticker symbol and filing date from the filing index (see *Chapter 16, Word Embeddings for Earnings Calls and SEC Filings*) to download daily adjusted stock prices for three months before and one month after the filing data as follows, capturing both the price data and unsuccessful tickers in the process:

```
yf_data, missing = [], []
for i, (symbol, dates) in enumerate(filing_index.groupby('ticker').date_filed,
                                    1):
    ticker = yf.Ticker(symbol)
    for idx, date in dates.to_dict().items():
        start = date - timedelta(days=93)
        end = date + timedelta(days=31)
        df = ticker.history(start=start, end=end)
        if df.empty:
            missing.append(symbol)
        else:
            yf_data.append(df.assign(ticker=symbol, filing=idx))
```

We obtain data on 3,954 tickers and source prices for a few hundred missing tickers using the Quandl Wiki data (see the notebook) and end up with 16,758 filings for 4,762 symbols.

Preprocessing SEC filing data

Compared to product reviews, financial text documents tend to be longer and have a more formal structure. In addition, in this case, we rely on data sourced from EDGAR that requires parsing of the XBRL source (see *Chapter 2, Market and Fundamental Data – Sources and Techniques*) and

may have errors such as including material other than the desired sections. We take several steps during preprocessing to address outliers and format the text data as integer sequences of equal length, as required by the model that we will build in the next section:

1. Remove all sentences that contain fewer than 5 or more than 50 tokens; this affects approximately. 5 percent of sentences.
2. Create 28,599 bigrams, 10,032 trigrams, and 2,372 n-grams with 4 elements.
3. Convert filings to a sequence of integers that represent the token frequency rank, removing filings with fewer than 100 tokens and truncating sequences at 20,000 elements.

Figure 19.16 highlights some corpus statistics for the remaining 16,538 filings with 179,214,369 tokens, around 204,206 of which are unique. The left panel shows the token frequency distribution on a log-log scale; the most frequent terms, "million," "business," "company," and "products" occur more than 1 million times each. As usual, there is a very long tail, with 60 percent of tokens occurring fewer than 25 times.

The central panel shows the distribution of the sentence lengths with a mode of around 10 tokens. Finally, the right panel shows the distribution of the filing length with a peak at 20,000 due to truncation:

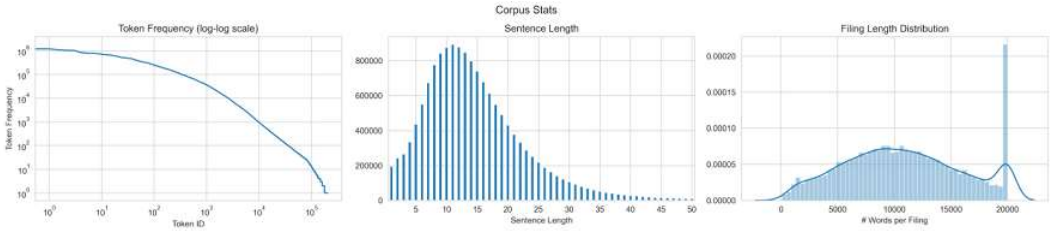


Figure 19.16: Cross-validation and test results for RNNs with multiple macro series

Preparing data for the RNN model

Now we need an outcome for our model to predict. We'll compute (somewhat arbitrarily) five-day forward returns for the day of filing (or the day

before if there are no prices for that date), assuming that filing occurred after market hours. Clearly, this assumption could be wrong, underscoring the need for **point-in-time data** emphasized in *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*, and *Chapter 3*, *Alternative Data for Finance – Categories and Use Cases*. We'll ignore this issue as the hidden cost of using free data.

We compute the forward returns as follows, removing outliers with weekly returns below 50 or above 100 percent:

```
fwd_return = {}
for filing in filings:
    date_filed = filing_index.at[filing, 'date_filed']
    price_data = prices[prices.filing==filing].close.sort_index()

    try:
        r = (price_data
             .pct_change(periods=5)
             .shift(-5)
             .loc[:date_filed]
             .iloc[-1])
    except:
        continue
    if not np.isnan(r) and -.5 < r < 1:
        fwd_return[filing] = r
```

This leaves us with 16,355 data points. Now we combine these outcomes with their matching filing sequences and convert the list of returns to a NumPy array:

```
y, X = [], []
for filing_id, fwd_ret in fwd_return.items():
    X.append(np.load(vector_path / f'{filing_id}.npy') + 2)
    y.append(fwd_ret)
y = np.array(y)
```

Finally, we create a 90:10 training/test split and use the `pad_sequences` function introduced in the first example in this section to generate fixed-

length sequences of 20,000 elements each:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.1)
X_train = pad_sequences(X_train,
                        truncating='pre',
                        padding='pre',
                        maxlen=maxlen)
X_test = pad_sequences(X_test,
                       truncating='pre',
                       padding='pre',
                       maxlen=maxlen)
X_train.shape, X_test.shape
((14719, 20000), (1636, 20000))
```

Building, training, and evaluating the RNN model

Now we can define our RNN architecture. The first layer learns the word embeddings. We define the embedding dimensions as previously, setting the following:

- The `input_dim` keyword to the size of the vocabulary
- The `output_dim` keyword to the size of each embedding
- The `input_length` parameter to how long each input sequence is going to be

For the recurrent layer, we use a bidirectional GRU unit that scans the text both forward and backward and concatenates the resulting output. We also add batch normalization and dropout for regularization with a five-unit dense layer before the linear output:

```
embedding_size = 100
input_dim = X_train.max() + 1
rnn = Sequential([
    Embedding(input_dim=input_dim,
              output_dim=embedding_size,
              input_length=maxlen,
             name='EMB'),
    BatchNormalization(name='BN1'),
    Bidirectional(GRU(32), name='BD1'),
```

```
    BatchNormalization(name='BN2'),
    Dropout(.1, name='DO1'),
    Dense(5, name='D'),
    Dense(1, activation='linear', name='OUT')])
```

The resulting model has over 2.5 million trainable parameters:

```
rnn.summary()
Layer (type)                 Output Shape              Param #
EMB (Embedding)              (None, 20000, 100)        2500000
BN1 (BatchNormalization)     (None, 20000, 100)        400
BD1 (Bidirectional)          (None, 64)                25728
BN2 (BatchNormalization)     (None, 64)                256
DO1 (Dropout)                (None, 64)                0
D (Dense)                    (None, 5)                 325
OUT (Dense)                  (None, 1)                 6
Total params: 2,526,715
Trainable params: 2,526,387
Non-trainable params: 328
```

We compile using the Adam optimizer, targeting the mean squared loss for this regression task while also tracking the square root of the loss and the mean absolute error as optional metrics:

```
rnn.compile(loss='mse',
            optimizer='Adam',
            metrics=[RootMeanSquaredError(name='RMSE'),
                     MeanAbsoluteError(name='MAE')])
```

With early stopping, we train for up to 100 epochs on batches of 32 observations each:

```
early_stopping = EarlyStopping(monitor='val_MAE',
                               patience=5,
                               restore_best_weights=True)
training = rnn.fit(X_train,
                   y_train,
                   batch_size=32,
```

```
                    epochs=100,
                    validation_data=(X_test, y_test),
                    callbacks=[early_stopping],
                    verbose=1)
```

The mean absolute error improves for only 4 epochs, as shown in the left panel of *Figure 19.17:*

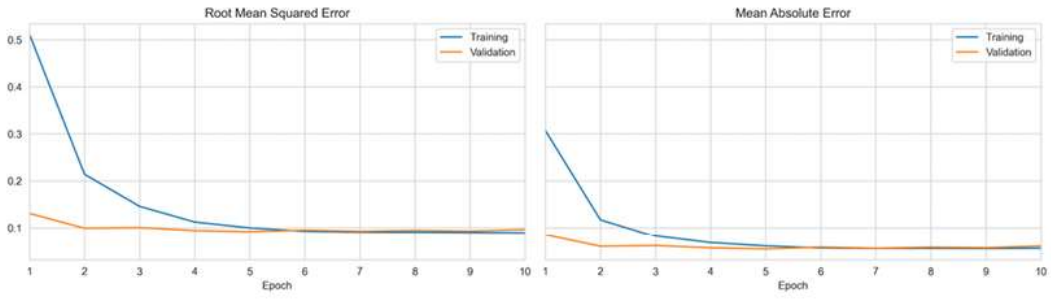


Figure 19.17: Cross-validation test results for RNNs using SEC filings to predict weekly returns

On the test set, the best model achieves a highly significant IC of 6.02:

```
y_score = rnn.predict(X_test)
rho, p = spearmanr(y_score.squeeze(), y_test)
print(f'{rho*100:.2f} ({p:.2%})')
6.02 (1.48%)
```

Lessons learned and next steps

The model is capable of generating return predictions that are significantly better than chance using only text data. There are both caveats that suggest taking the results with a grain of salt and reasons to believe we could improve on the result of this experiment.

On the one hand, the quality of both the stock price data and the parsed SEC filings is far from perfect. It's unclear whether price data issues bias the results positively or negatively, but they certainly increase the margin

of error. More careful parsing and cleaning of the SEC filings would most likely improve the results by removing noise.

On the other hand, there are numerous optimizations that may well improve the result. Starting with the text input, we did not attempt to parse the filing content beyond selecting certain sections; there may be value in removing boilerplate language or otherwise trying to pick the most meaningful statements. We also made somewhat arbitrary choices about the maximum length of filings and the size of the vocabulary that we could revisit. We could also shorten or lengthen the weekly prediction horizon. Furthermore, there are multiple aspects of the model architecture that we could refine, from the size of the embeddings to the number and size of layers and the degree of regularization.

Most fundamentally, we could combine the text input with a richer set of complementary features, as demonstrated in the previous section, using stacked LSTM with multiple inputs. Finally, we would certainly want a larger set of filings.

Summary

In this chapter, we presented the specialized RNN architecture that is tailored to sequential data. We covered how RNNs work, analyzed the 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 gated units like long short-term memory cells enable RNNs to learn dependencies over hundreds of time steps. Finally, we applied RNNs to challenges common in algorithmic trading, such as predicting univariate and multivariate time series and sentiment analysis using SEC filings.

In the next chapter, we will introduce unsupervised deep learning techniques like autoencoders and generative adversarial networks and their applications to investment and trading strategies.