18

CNNs for Financial Time Series and Satellite Images

In this chapter, we introduce the first of several specialized deep learning architectures that we will cover in *Part 4*. Deep **convolutional neural networks** (**CNNs**) have enabled superhuman performance in various computer vision tasks such as classifying images and video and detecting and recognizing objects in images. CNNs can also extract signals from time-series data that shares certain characteristics with image data and have been successfully applied to speech recognition (Abdel-Hamid et al. 2014). Moreover, they have been shown to deliver state-of-the-art performance on time-series classification across various domains (Ismail Fawaz et al. 2019).

CNNs are named after a linear algebra operation called a **convolution** that replaces the general matrix multiplication typical of feedforward networks (discussed in the last chapter) in at least one of their layers. We will show how convolutions work and why they are particularly well suited to data with a certain regular structure typically found in images but also present in time series.

Research into **CNN architectures** has proceeded very rapidly, and new architectures that improve benchmark performance continue to emerge. We will describe a set of building blocks consistently used by successful applications. We will also demonstrate how **transfer learning** can speed up learning by using pretrained weights for CNN layers closer to the input while fine-tuning the final layers to a specific task. We will also illustrate how to use CNNs for the specific computer vision task of **object detection**.

CNNs can help build a **trading strategy** by generating signals from images or (multiple) time-series data:

- **Satellite data** may signal future commodity trends, including the supply of certain crops or raw materials via aerial images of agricultural areas, mines, or transport networks like oil tankers. **Surveillance camera** footage, for example, from shopping malls, could be used to track and predict consumer activity.

- **Time-series data** encompasses a very broad range of data sources and CNNs have been shown to deliver high-quality classification results by exploiting their structural similarity with images.

We will create a trading strategy based on predictions of a CNN that uses time-series data that's been deliberately formatted like images and demonstrate how to build a CNN to classify satellite images.

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

- How CNNs employ several building blocks to efficiently model grid-like data
- Training, tuning, and regularizing CNNs for images and time-series data using TensorFlow
- Using transfer learning to streamline CNNs, even with less data
- Designing a trading strategy using return predictions by a CNN trained on time-series data formatted like images
- How to classify satellite images

You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

How CNNs learn to model grid-like data

CNNs are conceptually similar to feedforward **neural networks** (**NNs**): they consist of units with parameters called weights and biases, and the training process adjusts these parameters to optimize the network's output for a given input according to a loss function. They are most commonly used for classification. Each unit uses its parameters to apply a linear operation to the input data or activations received from other units, typically followed by a nonlinear transformation.

The overall network models a **differentiable function** that maps raw data, such as image pixels, to class probabilities using an output activation function like softmax. CNNs use an objective function such as cross-entropy loss to measure the quality of the output with a single metric. They also rely on the gradients of the loss with respect to the network parameter to learn via backpropagation.

Feedforward NNs with fully connected layers do not scale well to high-dimensional image data with a large number of pixel values. Even the low-resolution images included in the CIFAR-10 dataset that we'll use in the next section contain 32×32 pixels with up to 256 different color values represented by 8 bits each. With three channels, for example, for the red, green, and blue channels of the RGB color model, a single unit in a fully connected input layer implies $32 \times 32 \times 3 = 3{,}072$ weights. A more standard resolution of 640×480 pixels already yields closer to 1 million weights for a single input unit. Deep architectures with several layers of meaningful width quickly lead to an exploding number of parameters that make overfitting during training all but certain.

A fully connected feedforward NN makes no assumptions about the local structure of the input data so that arbitrarily reordering the features has no impact on the training result. By contrast, CNNs make the **key assumption** that the **data has a grid-like topology** and that the **local structure matters**. In other words, they encode the assumption that the input has a structure typically found in image data: pixels form a two-dimensional grid, possibly with several channels to represent the components of the color signal. Furthermore, the values of nearby pixels are likely more relevant to detect key features such as edges and corners than faraway data points. Naturally, initial CNN applications such as handwriting recognition focused on image data.

Over time, however, researchers recognized **similar characteristics in time-series data**, broadening the scope for the productive use of CNNs. Time-series data consists of measurements at regular intervals that create a one-dimensional grid along the time axis, such as the lagged returns for a given ticker. There can also be a second dimension with additional features for this ticker and the same time periods. Finally, we could represent additional tickers using the third dimension.

A common CNN use case beyond images includes audio data, either in a one-dimensional waveform in the time domain or, after a Fourier transform, as a two-dimensional spectrum in the frequency domain. CNNs also play a key role in AlphaGo, the first algorithm to win a game of Go against humans, where they evaluated different positions on the grid-like board.

The most important element to encode the **assumption of a grid-like topology** is the **convolution** operation that gives CNNs their name, combined with **pooling**. We will see that the specific assumptions about the functional relationship between input and output data imply that CNNs need far fewer parameters and compute more efficiently.

In this section, we will explain how convolution and pooling layers learn filters that extract local features and why these operations are particularly suitable for data with the structure just described. State-of-the-art CNNs combine many of these basic building blocks to achieve the layered representation learning described in the previous chapter. We conclude by describing key architectural innovations over the last decade that saw enormous performance improvements.

From hand-coding to learning filters from data

For image data, this local structure has traditionally motivated the development of hand-coded filters that extract such patterns for the use as features in **machine learning** (**ML**) models.

Figure 18.1 displays the effect of simple filters designed to detect certain edges. The notebook `filter_example.ipynb` illustrates how to use hand-coded filters in a convolutional network and visualizes the resulting transformation of the image. The filters are simple [-1, 1] patterns arranged in a 2×2 matrix, shown in the upper right of the figure. Below each filter, its effects are shown; they are a bit subtle and will be easier to spot in the accompanying notebook.

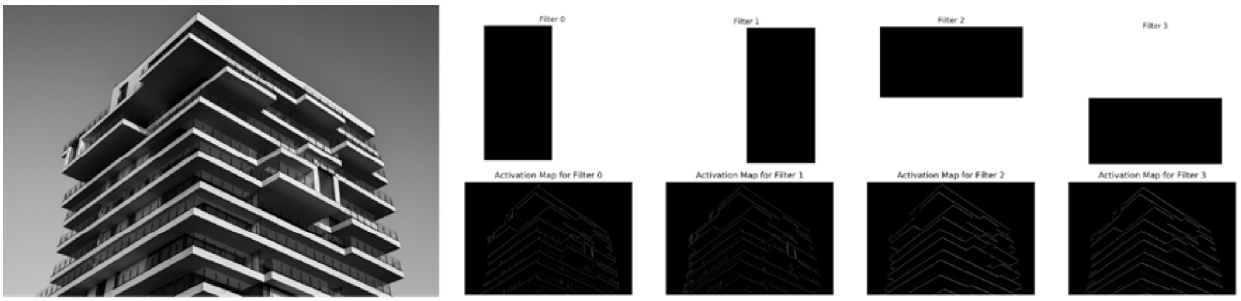


Figure 18.1: The result of basic edge filters applied to an image

Convolutional layers, by contrast, are designed to learn such local feature representations from the data. A key insight is to restrict their input, called the **receptive field**, to a small area of the input so it captures basic pixel constellations that reflect common patterns like edges or corners. Such patterns may occur anywhere in an image, though, so CNNs also need to recognize similar patterns in different locations and possibly with small variations.

Subsequent layers then learn to synthesize these local features to detect **higher-order features**. The linked resources on GitHub include examples of how to visualize the filters learned by a deep CNN using some of the deep architectures that we present in the next section on reference architectures.

How the elements of a convolutional layer operate

Convolutional layers integrate **three architectural ideas** that enable the learning of feature representations that are to some degree invariant to shifts, changes in scale, and distortion:

- Sparse rather than dense connectivity
- Weight sharing
- Spatial or temporal downsampling

Moreover, convolutional layers allow for inputs of variable size. We will walk through a typical convolutional layer and describe each of these ideas in turn.

Figure 18.2 outlines the set of operations that typically takes place in a three-dimensional convolutional layer, assuming image data is input with the three dimensions of height, width, and depth, or the number of channels. The range of pixel values depends on the bit representation, for example, [0, 255] for 8 bits. Alternatively, the width axis could represent time, the height different features, and the channels could capture observations on distinct objects such as tickers.

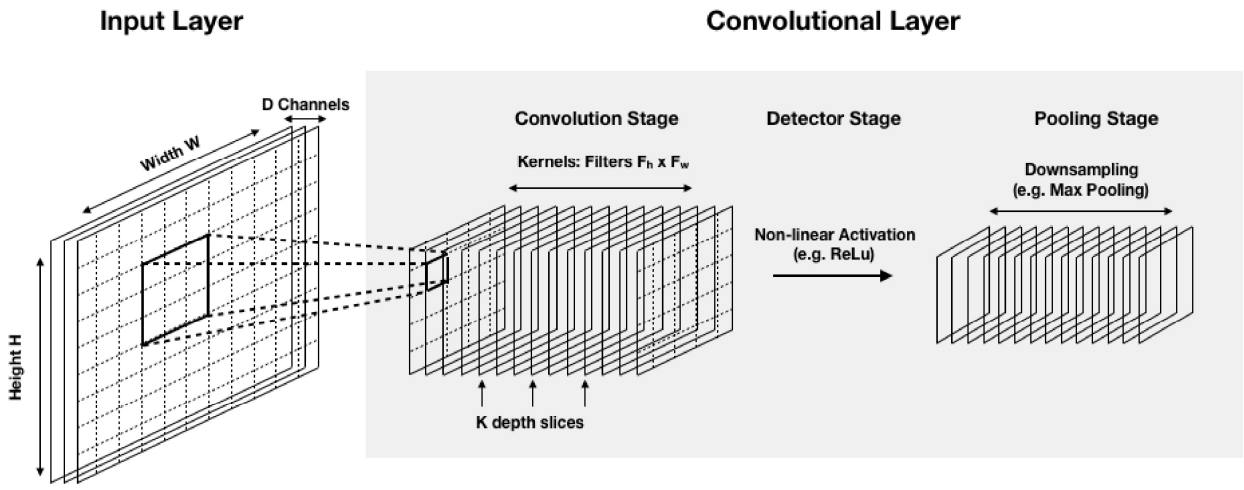


Figure 18.2: Typical operations in a two-dimensional convolutional layer

Successive computations process the input through the convolutional, detector, and pooling stages that we describe in the next three sections. In the example depicted in *Figure 18.2*, the convolutional layer receives three-dimensional input and produces an output of the same dimensionality.

State-of-the-art CNNs are composed of several such layers of varying sizes that are either stacked on top of each other or operate in parallel on different branches. With each layer, the network can detect higher-level, more abstract features.

The convolution stage – extracting local features

The first stage applies a filter, also called the **kernel**, to overlapping patches of the input image. The filter is a matrix of a much smaller size than the input so that its receptive field is limited to a few contiguous values such as pixels or time-series values. As a result, it focuses on local patterns and dramatically reduces the number of parameters and computations relative to a fully connected layer.

A complete convolutional layer has several **feature maps** organized as depth slices (depicted in *Figure 18.2*) so that each layer can extract multiple features.

From filters to feature maps

While scanning the input, the kernel is convolved with each input segment covered by its receptive field. The convolution operation is simply the dot product between the filter weights and the values of the matching input area after both have been reshaped to vectors. Each convolution thus produces a single number, and the entire scan yields a feature map. Since the dot product is maximized for identical vectors, the feature map indicates the degree of activation for each input region.

Figure 18.3 illustrates the result of the scan of a 5 × 5 input using a 3 × 3 filter with given values, and how the activation in the upper-right corner of the feature map results from the dot product of the flattened input region and the kernel:

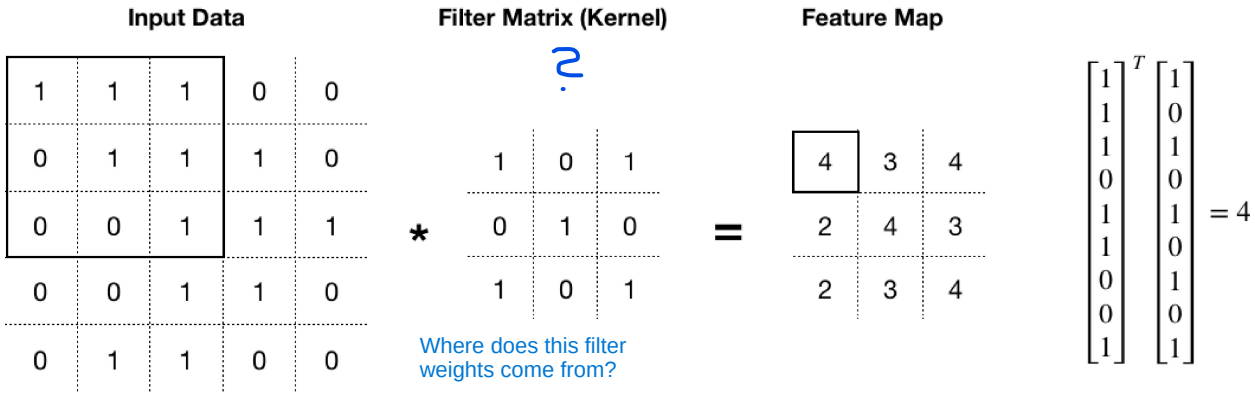


Figure 18.3: From convolutions to a feature map

The most important aspect is that the **filter values are the parameters** of the convolutional layers, **learned from the data** during training to minimize the chosen loss function. In other words, CNNs learn useful feature representations by finding kernel values that activate input patterns that are most useful for the task at hand.

How to scan the input – strides and padding

The **stride** defines the step size used for scanning the input, that is, the number of pixels to shift horizontally and vertically. Smaller strides scan more (overlapping) areas but are computationally more expensive. Four options are commonly used when the filter does not fit the input perfectly and partially crosses the image boundary during the scan:

- **Valid convolution**: Discards scans where the image and filter do not perfectly match
- **Same convolution**: Zero-pads the input to produce a feature map of equal size
- **Full convolution**: Zero-pads the input so that each pixel is scanned an equal number of times, including pixels at the border (to avoid oversampling pixels closer to the center)
- **Causal**: Zero-pads the input only on the left so that the output does not depend on an input from a later period; maintains the temporal order for time-series data

The choices depend on the nature of the data and where useful features are most likely located. In combination with the number of depth slices, they determine the output size of the convolution stage. The Stanford lecture notes by Andrew Karpathy (see GitHub) contain helpful examples using NumPy.

Parameter sharing for robust features and fast computation

The location of salient features may vary due to distortion or shifts. Furthermore, elementary feature detectors are likely useful across the entire image. CNNs encode these assumptions by sharing or tying the weights for the filter in a given depth slice.

As a result, each depth slice specializes in a certain pattern and the number of parameters is further reduced. Weight sharing works less well, however, when images are spatially centered and key patterns are less likely to be uniformly distributed across the input area.

The detector stage – adding nonlinearity

The feature maps are usually passed through a nonlinear transformation. The **rectified linear unit** (**ReLU**) that we encountered in the last chapter is a common function for this purpose. ReLUs replace negative activations element-wise by zero and mitigate the risk of vanishing gradients found in other activation functions such as tanh (see *Chapter 17*, *Deep Learning for Trading*).

A popular alternative is the **softplus function**:

$$f(x) = \ln(1 + e^x)$$

In contrast to ReLU, it has a derivative everywhere, namely the sigmoid function that we used for logistic regression (see *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*).

The pooling stage – downsampling the feature maps

The last stage of the convolutional layer may downsample the feature map's input representation to do the following:

- Reduce its dimensionality and prevent overfitting
- Lower the computational cost
- Enable basic translation invariance

This assumes that the precise location of the features is not only less important for identifying a pattern but can even be harmful because it will likely vary for different instances of the target. Pooling lowers the spatial resolution of the feature map as a simple way to render the location information less precise. However, this step is optional and many architectures use pooling only for some layers or not at all.

A common pooling operation is **max pooling**, which uses only the maximum activation value from (typically) non-overlapping subregions. For a small 4×4 feature map, for instance, 2×2 max pooling outputs the maximum for each of the four non-overlapping 2×2 areas. Less common pooling operators use the average or the median. Pooling does not add or learn new parameters but the size of the input window and possibly the stride are additional hyperparameters.

The evolution of CNN architectures – key innovations

Several CNN architectures have pushed performance boundaries over the past two decades by introducing important innovations. Predictive performance growth accelerated dramatically with the arrival of big data in the form of ImageNet (Fei-Fei 2015) with 14 million images assigned to 20,000 classes by humans via Amazon's Mechanical Turk. The **ImageNet Large Scale Visual Recognition Challenge** (**ILSVRC**) became the focal point of CNN progress around a slightly smaller set of 1.2 million images from 1,000 classes.

It is useful to be familiar with the **reference architectures** dominating these competitions for practical reasons. As we will see in the next section on working with CNNs for image data, they offer a good starting point for standard tasks. Moreover, **transfer learning** allows us to address many computer vision tasks by building on a successful architecture with pretrained weights. Transfer learning not only speeds up architecture selection and training but also enables successful applications on much smaller datasets.

In addition, many publications refer to these architectures, and they often serve as a basis for networks tailored to segmentation or localization tasks. We will further describe some landmark architectures in the section on image classification and transfer learning.

Performance breakthroughs and network size

The left side of *Figure 18.4* plots the top-1 accuracy against the computational cost of a variety of network architectures. It suggests a positive relationship between the number of parameters and performance, but also shows that the marginal benefit of more parameters declines and that architectural design and innovation also matter.

The right side plots the top-1 accuracy per parameter for all networks. Several new architectures target use cases on less powerful devices such as mobile phones. While they do not achieve state-of-the-art performance, they have found much more efficient implementations. See the resources on GitHub for more details on these architectures and the analysis behind these charts.

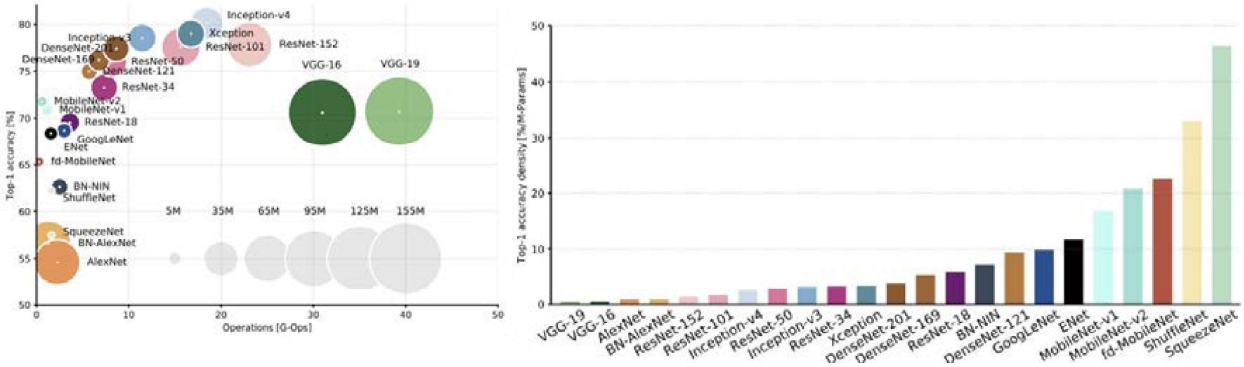


Figure 18.4: Predictive performance and computational complexity

Lessons learned

Some of the lessons learned from 20 years of CNN architecture developments, especially since 2012, include the following:

- **Smaller convolutional** filters perform better (possibly except at the first layer) because several small filters can substitute for a larger filter at a lower computational cost.
- **1 × 1 convolutions** reduce the dimensionality of feature maps so that the network can learn a larger number overall.
- **Skip connections** are able to create multiple paths through the network and enable the training of much higher-capacity CNNs.

CNNs for satellite images and object detection

In this section, we demonstrate how to solve key computer vision tasks such as image classification and object detection. As mentioned in the introduction and in *Chapter 3*, *Alternative Data for Finance – Categories and Use Cases*, image data can inform a trading strategy by providing clues about future trends, changing fundamentals, or specific events relevant to a target asset class or investment universe. Popular examples include exploiting satellite images for clues about the supply of agricultural commodities, consumer and economic activity, or the status of manufacturing or raw material supply chains. Specific tasks might include the following, for example:

- **Image classification**: Identifying whether cultivated land for certain crops is expanding, or predicting harvest quality and quantities
- **Object detection**: Counting the number of oil tankers on a certain transport route or the number of cars in a parking lot, or identifying the locations of shoppers in a mall

In this section, we'll demonstrate how to design CNNs to automate the extraction of such information, both from scratch using popular architectures and via transfer learning that fine-tunes pretrained weights to a given task. We'll also demonstrate how to detect objects in a given scene.

We will introduce key CNN architectures for these tasks, explain why they work well, and show how to train them using TensorFlow 2. We will also demonstrate how to source pretrained weights and fine-tune time. Unfortunately, satellite images with information directly relevant for a trading strategy are very costly to obtain and are not readily available. We will, however, demonstrate how to work with the EuroSat dataset to build a classifier that identifies different land uses. This brief introduction to CNNs for computer vision aims to demonstrate how to approach common tasks that you will likely need to tackle when aiming to design a trading strategy based on images relevant to the investment universe of your choice.

All the libraries we introduced in the last chapter provide support for convolutional layers; we'll focus on the Keras interface of TensorFlow 2. We are first going to illustrate the LeNet5 architecture using the MNIST handwritten digit dataset. Next, we'll demonstrate the use of data augmentation with AlexNet on CIFAR-10, a simplified version of the original ImageNet. Then we'll continue with transfer learning based on state-of-the-art architectures before we apply what we've learned to actual satellite images. We conclude with an example of object detection in real-life scenes.

LeNet5 – The first CNN with industrial applications

Yann LeCun, now the Director of AI Research at Facebook, was a leading pioneer in CNN development. In 1998, after several iterations starting in the 1980s, LeNet5 became the first modern CNN used in real-world applications that introduced several architectural elements still relevant today.

LeNet5 was published in a very instructive paper, *Gradient-Based Learning Applied to Document Recognition* (LeCun et al. 1989), that laid out many of the central concepts. Most importantly, it promoted the insight that convolutions with learnable filters are effective at extracting related features at multiple locations with few parameters. Given the limited computational resources at the time, efficiency was of paramount importance.

LeNet5 was designed to recognize the handwriting on checks and was used by several banks. It established a new benchmark for classification accuracy, with a result of 99.2 percent on the MNIST handwritten digit dataset. It consists of three convolutional layers, each containing a nonlinear tanh transformation, a pooling operation, and a fully connected output layer. Throughout the convolutional layers, the number of feature maps increases while their dimensions decrease. It has a total of 60,850 trainable parameters (Lecun et al. 1998).

"Hello World" for CNNs – handwritten digit classification

In this section, we'll implement a slightly simplified version of LeNet5 to demonstrate how to build a CNN using a TensorFlow implementation. The original MNIST dataset contains 60,000 grayscale images in 28 × 28 pixel resolution, each containing a single handwritten digit from 0 to 9. A good alternative is the more challenging but structurally similar Fashion MNIST dataset that we encountered in *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*. See the `digit_classification_with_lenet5` notebook for implementation details.

We can load it in Keras out of the box:

```
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train.shape, X_test.shape
((60000, 28, 28), (10000, 28, 28))
```

Figure 18.5 shows the first ten images in the dataset and highlights significant variation among instances of the same digit. On the right, it shows how the pixel values for an individual image range from 0 to 255:

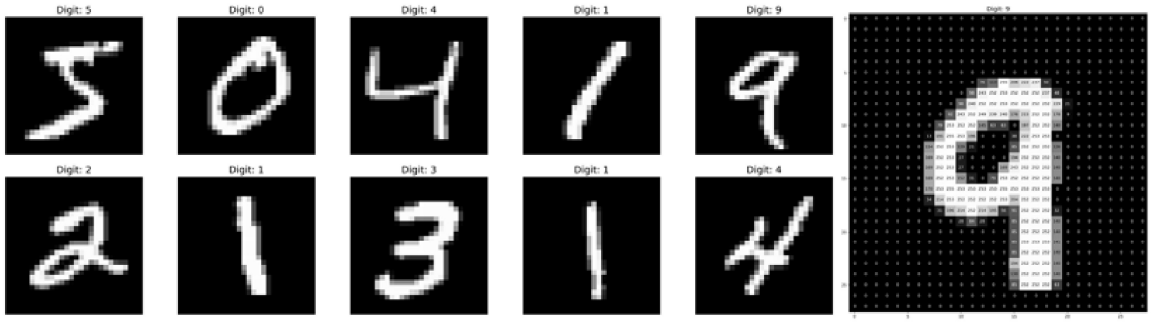


Figure 18.5: MNIST sample images

We rescale the pixel values to the range [0, 1] to normalize the training data and facilitate the backpropagation process and convert the data to 32-bit floats, which reduce memory requirements and computational cost while providing sufficient precision for our use case:

```
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
```

Defining the LeNet5 architecture

We can define a simplified version of LeNet5 that omits the original final layer containing radial basis functions as follows, using the default "valid" padding and single-step strides unless defined otherwise:

```
lenet5 = Sequential([
    Conv2D(filters=6, kernel_size=5, activation='relu',
           input_shape=(28, 28, 1), name='CONV1'),
    AveragePooling2D(pool_size=(2, 2), strides=(1, 1),
                     padding='valid', name='POOL1'),
    Conv2D(filters=16, kernel_size=(5, 5), activation='tanh', name='CONV2'),
    AveragePooling2D(pool_size=(2, 2), strides=(2, 2), name='POOL2'),
    Conv2D(filters=120, kernel_size=(5, 5), activation='tanh', name='CONV3'),
    Flatten(name='FLAT'),
    Dense(units=84, activation='tanh', name='FC6'),
    Dense(units=10, activation='softmax', name='FC7')
])
```

The summary indicates that the model thus defined has over 300,000 parameters:

```
Layer (type)                 Output Shape              Param #
CONV1 (Conv2D)               (None, 24, 24, 6)         156
POOL1 (AveragePooling2D)     (None, 23, 23, 6)         0
CONV2 (Conv2D)               (None, 19, 19, 16)        2416
_________________________________________________________________
POOL2 (AveragePooling2D)     (None, 9, 9, 16)          0
_________________________________________________________________
CONV3 (Conv2D)               (None, 5, 5, 120)         48120
```

```
_________________________________________________________________
FLAT (Flatten)               (None, 3000)              0
_________________________________________________________________
FC6 (Dense)                  (None, 84)                252084
_________________________________________________________________
FC7 (Dense)                  (None, 10)                850
=================================================================
Total params: 303,626
Trainable params: 303,626
```

We compile with `sparse_crossentropy_loss`, which accepts integers rather than one-hot-encoded labels and the original stochastic gradient optimizer:

```
lenet5.compile(loss='sparse_categorical_crossentropy',
               optimizer='SGD',
               metrics=['accuracy'])
```

Training and evaluating the model

Now we are ready to train the model. The model expects four-dimensional input, so we reshape accordingly. We use the standard batch size of 32 and an 80:20 train-validation split. Furthermore, we leverage checkpointing to store the model weights if the validation error improves, and make sure the dataset is randomly shuffled. We also define an `early_stopping` callback to interrupt training once the validation accuracy no longer improves for 20 iterations:

```
lenet_history = lenet5.fit(X_train.reshape(-1, 28, 28, 1),
                           y_train,
                           batch_size=32,
                           epochs=100,
                           validation_split=0.2, # use 0 to train on all data
                           callbacks=[checkpointer, early_stopping],
                           verbose=1,
                           shuffle=True)
```

The training history records the last improvement after 81 epochs that take around 4 minutes on a single GPU. The test accuracy of this sample run is 99.09 percent, almost exactly the same result as for the original LeNet5:

```
accuracy = lenet5.evaluate(X_test.reshape(-1, 28, 28, 1), y_test, verbose=0)
[1]
print('Test accuracy: {:.2%}'.format(accuracy))
Test accuracy: 99.09%
```

For comparison, a simple two-layer feedforward network achieves "only" 97.04 percent test accuracy (see the notebook). The LeNet5 improvement on MNIST is, in fact, modest. Non-neural methods have also achieved classification accuracies greater than or equal to 99 percent, including K-nearest neighbors and support vector machines. CNNs really shine with more challenging datasets as we will see next.

AlexNet – reigniting deep learning research

AlexNet, developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton at the University of Toronto, dramatically reduced the error rate and significantly outperformed the runner-up at the 2012 ILSVRC, achieving a top-5 error of 16 percent versus 26 percent (Krizhevsky, Sutskever, and Hinton 2012). This breakthrough triggered a renaissance in ML research and put deep learning for computer vision firmly on the global technology map.

The AlexNet architecture is similar to LeNet, but much deeper and wider. It is often credited with discovering **the importance of depth** with around 60 million parameters, exceeding LeNet5 by a factor of 1,000, a testament to increased computing power, especially the use of GPUs, and much larger datasets.

It included convolutions stacked on top of each other rather than combining each convolution with a pooling stage, and successfully used dropout for regularization and ReLU for efficient nonlinear transformations. It also employed data augmentation to increase the number of training samples, added weight decay, and used a more efficient implementation of convolutions. It also accelerated training by distributing the network over two GPUs.

The notebook `image_classification_with_alexnet.ipynb` has a slightly simplified version of AlexNet tailored to the CIFAR-10 dataset that contains 60,000 images from 10 of the original 1,000 classes. It has been compressed to a 32×32 pixel resolution from the original 224×224, but still has three color channels.

See the notebook `image_classification_with_alexnet` for implementation details; we will skip over some repetitive steps here.

Preprocessing CIFAR-10 data using image augmentation

CIFAR-10 can also be downloaded using TensorFlow's Keras interface, and we rescale the pixel values and one-hot encode the ten class labels as we did with MNIST in the previous section.

We first train a two-layer feedforward network on 50,000 training samples for 45 epochs to achieve a test accuracy of 45.78 percent. We also experiment with a three-layer convolutional net with over 528,000 parameters that achieves 74.51 percent test accuracy (see the notebook).

A common trick to enhance performance is to artificially increase the size of the training set by creating synthetic data. This involves randomly shifting or horizontally flipping the image or introducing noise into the image. TensorFlow includes an `ImageDataGenerator` class for this purpose. We can configure it and fit the training data as follows:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
    width_shift_range=0.1,   # randomly horizontal shift
    height_shift_range=0.1,  # randomly vertical shift
    horizontal_flip=True)    # randomly horizontal flip
datagen.fit(X_train)
```

The result shows how the augmented images (in low 32×32 resolution) have been altered in various ways as expected:

Figure 18.6: Original and augmented samples

The test accuracy for the three-layer CNN improves modestly to 76.71 percent after training on the larger, augmented data.

Defining the model architecture

We need to adapt the AlexNet architecture to the lower dimensionality of CIFAR-10 images relative to the ImageNet samples used in the competition. To this end, we use the original number of filters but make them smaller (see the notebook for implementation details).

The summary (see the notebook) shows the five convolutional layers followed by two fully connected layers with frequent use of batch normalization, for a total of 21.5 million parameters.

Comparing AlexNet performance

In addition to AlexNet, we trained a 2-layer feedforward NN and a 3-layer CNN, the latter with and without image augmentation. After 100 epochs (with early stopping if the validation accuracy does not improve for 20 rounds), we obtain the cross-validation trajectories and test accuracy for the four models, as displayed in *Figure 18.7*:

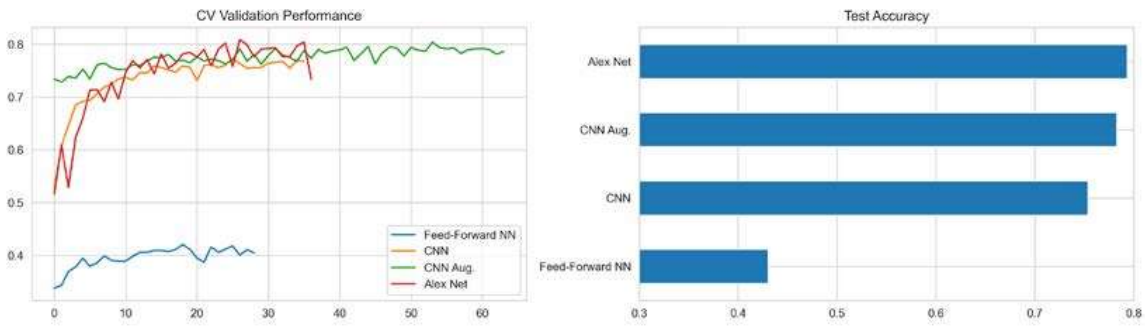


Figure 18.7: Validation performance and test accuracy on CIFAR-10

AlexNet achieves the highest test accuracy with 79.33 percent after some 35 epochs, closely followed by the shallower CNN with augmented images at 78.29 percent that trains for longer due to the larger dataset. The feedforward NN performs much worse than on MNIST on this more complex dataset, with a test accuracy of 43.05 percent.

Transfer learning – faster training with less data

In practice, sometimes we do not have enough data to train a CNN from scratch with random initialization. **Transfer learning** is an ML technique that repurposes a model trained on one set of data for another task. Naturally, it works if the learning from the first task carries over to the task of interest. If successful, it can lead to better performance and faster training that requires less labeled data than training a neural network from scratch on the target task.

Alternative approaches to transfer learning

The transfer learning approach to CNN relies on pretraining on a very large dataset like ImageNet. The goal is for the convolutional filters to extract a feature representation that generalizes to new images. In a second step, it leverages the result to either initialize and retrain a new CNN or use it as input to a new network that tackles the task of interest.

As discussed, CNN architectures typically use a sequence of convolutional layers to detect hierarchical patterns, adding one or more fully connected layers to map the convolutional activations to the outcome classes or values. The output of the last convolutional layer that feeds into the fully connected part is called the bottleneck features. We can use the **bottleneck features** of a pretrained network as inputs into a new fully connected network, usually after applying a ReLU activation function.

In other words, we freeze the convolutional layers and **replace the dense part of the network**. An additional benefit is that we can then use inputs of different sizes because it is the dense layers that constrain the input size.

Alternatively, we can use the bottleneck features as **inputs into a different machine learning algorithm**. In the AlexNet architecture, for instance, the bottleneck layer computes a vector with 4,096 entries for each 224×224 input image. We then use this vector as features for a new model.

We also can go a step further and not only replace and retrain the final layers using new data but also **fine-tune the weights of the pretrained CNN**. To achieve this, we continue training, either only for later layers while freezing the weights of some earlier layers, or for all layers. The motivation is presumably to preserve more generic patterns learned by lower layers, such as edge or color blob detectors, while allowing later layers of the CNN to adapt to the details of a new task. ImageNet, for example, contains a wide variety of dog breeds, which may lead to feature representations specifically useful for differentiating between these classes.

Building on state-of-the-art architectures

Transfer learning permits us to leverage top-performing architectures without incurring the potentially fairly GPU- and data-intensive training. We briefly outline the key characteristics of a few additional popular architectures that are popular starting points.

VGGNet – more depth and smaller filters

The runner-up in ILSVRC 2014 was developed by Oxford University's Visual Geometry Group (VGG, Simonyan 2015). It demonstrated the effectiveness of **much smaller** 3×3 **convolutional filters** combined in sequence and reinforced the importance of depth for strong performance. VGG16 contains 16 convolutional and fully connected layers that only perform 3×3 convolutions and 2×2 pooling (see *Figure 18.5*).

VGG16 has **140 million parameters** that increase the computational costs of training and inference as well as the memory requirements. However, most parameters are in the fully connected layers that were since discovered not to be essential so that removing them greatly reduces the number of parameters without negatively impacting performance.

GoogLeNet – fewer parameters through Inception

Christian Szegedy at Google reduced the computational costs using more efficient CNN implementations to facilitate practical applications at scale. The resulting GoogLeNet (Szegedy et al. 2015) won the ILSVRC 2014 with only 4 million parameters due to the **Inception module**, compared to AlexNet's 60 million and VGG16's 140 million.

The Inception module builds on the **network-in-network concept** that uses 1×1 convolutions to compress a deep stack of convolutional filters and thus reduce the cost of computation. The module uses parallel 1×1, 3×3, and 5×5 filters, combining the latter two with 1×1 convolutions to reduce the dimensionality of the filters passed in by the previous layer.

In addition, it uses average pooling instead of fully connected layers on top of the convolutional layers to eliminate many of the less impactful parameters. There have been several enhanced versions, most recently Inception-v4.

ResNet – shortcut connections beyond human performance

The **residual network (ResNet)** architecture was developed at Microsoft and won the ILSVRC 2015. It pushed the top-5 error to 3.7 percent, below the level of human performance on this task of around 5 percent (He et al. 2015).

It introduces identity shortcut connections that skip several layers and overcome some of the challenges of training deep networks, enabling the use of hundreds or even over a thousand layers. It also heavily uses batch normalization, which was shown to allow higher learning rates and be more forgiving about weight initialization. The architecture also omits the fully connected final layers.

As mentioned in the last chapter, the training of deep networks faces the notorious vanishing gradient challenge: as the gradient propagates to earlier layers, repeated multiplication of small weights risks shrinking the gradient toward zero. Hence, increasing depth may limit learning.

The shortcut connection that skips two or more layers has become one of the most popular developments in CNN architectures and triggered numerous research efforts to refine and explain its performance. See the references on GitHub for additional information.

Transfer learning with VGG16 in practice

Modern CNNs can take weeks to train on multiple GPUs on ImageNet, but fortunately, many researchers share their final weights. TensorFlow 2, for example, contains pretrained models for several of the reference architectures discussed previously, namely VGG16 and its larger version, VGG19, ResNet50, InceptionV3, and InceptionResNetV2, as well as MobileNet, DenseNet, NASNet, and MobileNetV2.

How to extract bottleneck features

The notebook `bottleneck_features.ipynb` illustrates how to download the pretrained VGG16 model, either with the final layers to generate predictions or without the final layers, as illustrated in *Figure 18.8*, to extract the outputs produced by the bottleneck features:

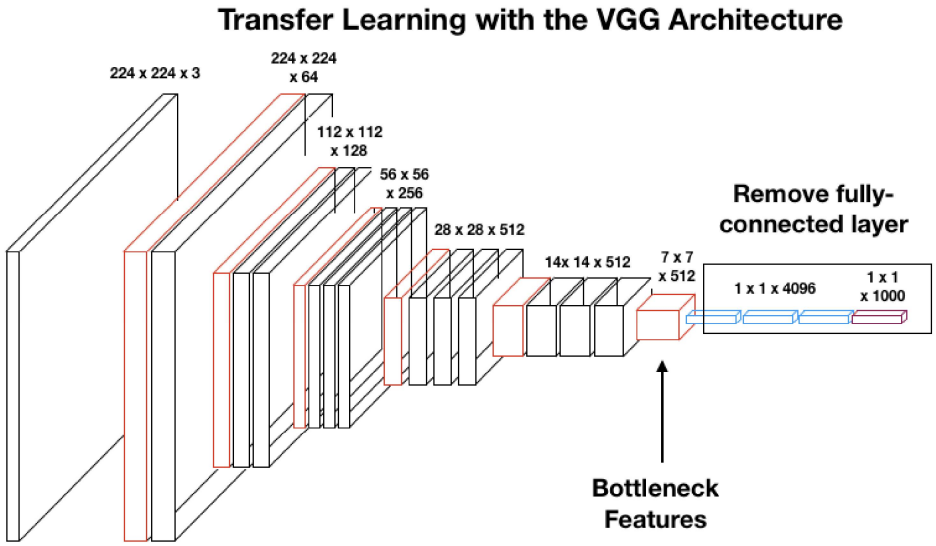


Figure 18.8: The VGG16 architecture

TensorFlow 2 makes it very straightforward to download and use pretrained models:

```
from tensorflow.keras.applications.vgg16 import VGG16
vgg16 = VGG16()
vgg16.summary()
Layer (type)                 Output Shape              Param #
input_1 (InputLayer)         (None, 224, 224, 3)       0
… several layers omitted...
```

```
block5_conv4 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
fc1 (Dense)                  (None, 4096)              102764544
fc2 (Dense)                  (None, 4096)              16781312
predictions (Dense)          (None, 1000)              4097000
Total params: 138,357,544
Trainable params: 138,357,544
```

You can use this model for predictions like any other Keras model: we pass in seven sample images and obtain class probabilities for each of the 1,000 ImageNet categories:

```
y_pred = vgg16.predict(img_input)
Y_pred.shape
(7, 1000)
```

To exclude the fully connected layers, just add the keyword `include_top=False`. Predictions are now output by the final convolutional layer `block5_pool` and match this layer's shape:

```
vgg16 = VGG16(include_top=False)
vgg16.predict(img_input).shape
(7, 7, 7, 512)
```

By omitting the fully connected layers and keeping only the convolutional modules, we are no longer forced to use a fixed input size for the model such as the original 224×224 ImageNet format. Instead, we can adapt the model to arbitrary input sizes.

How to fine-tune a pretrained model

We will demonstrate how to freeze some or all of the layers of a pretrained model and continue training using a new fully-connected set of layers and data with a different format (see the notebook `transfer_learning.ipynb` for code examples, adapted from a TensorFlow 2 tutorial).

We use the VGG16 weights, pretrained on ImageNet with TensorFlow's built-in cats versus dogs images (see the notebook on how to source the dataset).

Preprocessing resizes all images to 160×160 pixels. We indicate the new input size as we instantiate the pretrained VGG16 instance and then freeze all weights:

```
vgg16 = VGG16(input_shape=IMG_SHAPE, include_top=False, weights='imagenet')
vgg16.trainable = False
vgg16.summary()
Layer (type)                 Output Shape              Param #
... omitted layers...
block5_conv3 (Conv2D)        (None, 10, 10, 512)         2359808
block5_pool (MaxPooling2D)   (None, 5, 5, 512)         0
```

```
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688
```

The shape of the model output for 32 sample images now matches that of the last convolutional layer in the headless model:

```
feature_batch = vgg16(image_batch)
Feature_batch.shape
TensorShape([32, 5, 5, 512])
```

We can append new layers to the headless model using either the Sequential or the Functional API. For the Sequential API, adding `GlobalAveragePooling2D`, `Dense`, and `Dropout` layers works as follows:

```
global_average_layer = GlobalAveragePooling2D()
dense_layer = Dense(64, activation='relu')
dropout = Dropout(0.5)
prediction_layer = Dense(1, activation='sigmoid')
seq_model = tf.keras.Sequential([vgg16,
                                 global_average_layer,
                                 dense_layer,
                                 dropout,
                                 prediction_layer])
seq_model.compile(loss = tf.keras.losses.BinaryCrossentropy(from_logits=True),
                        optimizer = 'Adam',
                        metrics=["accuracy"])
```

We set `from_logits=True` for the `BinaryCrossentropy` loss because the model provides a linear output. The summary shows how the new model combines the pretrained VGG16 convolutional layers and the new final layers:

```
seq_model.summary()
Layer (type)                 Output Shape              Param #
vgg16 (Model)                (None, 5, 5, 512)         14714688
global_average_pooling2d (Gl (None, 512)               0
dense_7 (Dense)              (None, 64)                32832
dropout_3 (Dropout)          (None, 64)                0
dense_8 (Dense)              (None, 1)                 65
Total params: 14,747,585
Trainable params: 11,831,937
Non-trainable params: 2,915,648
```

See the notebook for the Functional API version.

Prior to training the new final layer, the pretrained VGG16 delivers a validation accuracy of 48.75 percent. Now we proceed to train the model for 10 epochs as follows, adjusting only the final layer weights:

```
history = transfer_model.fit(train_batches,
                             epochs=initial_epochs,
                             validation_data=validation_batches)
```

10 epochs boost validation accuracy above 94 percent. To fine-tune the model, we can unfreeze the VGG16 models and continue training. Note that you should only do so after training the new final layers: randomly initialized classification layers will likely produce large gradient updates that can eliminate the pretraining results.

To unfreeze parts of the model, we select a layer, after which we set the weights to `trainable`; in this case, layer 12 of the total 19 layers in the VGG16 architecture:

```
vgg16.trainable = True
len(vgg16.layers)
19
# Fine-tune from this layer onward
start_fine_tuning_at = 12
# Freeze all the layers before the 'fine_tune_at' layer
for layer in vgg16.layers[:start_fine_tuning_at]:
    layer.trainable =  False
```

Now just recompile the model and continue training for up to 50 epochs using early stopping, starting in epoch 10 as follows:

```
fine_tune_epochs = 50
total_epochs = initial_epochs + fine_tune_epochs
history_fine_tune = transfer_model.fit(train_batches,
                                       epochs=total_epochs,
                                       initial_epoch=history.epoch[-1],
                                       validation_data=validation_batches,
                                       callbacks=[early_stopping])
```

Figure 18.9 shows how the validation accuracy increases substantially, reaching 97.89 percent after another 22 epochs:

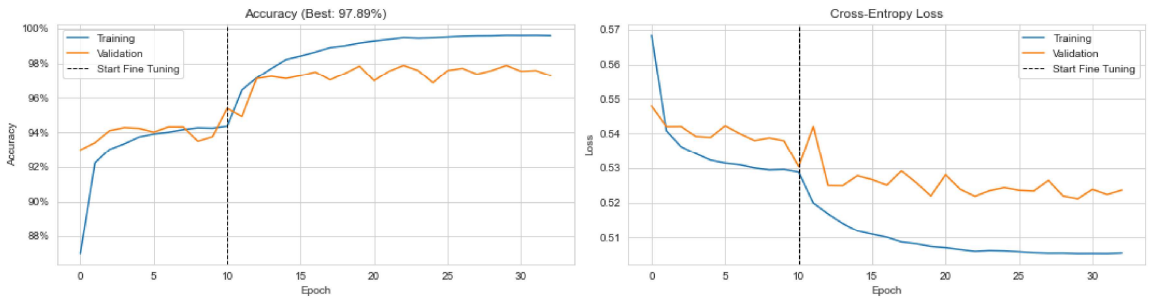


Figure 18.9: Cross-validation performance: accuracy and cross-entropy loss

Transfer learning is an important technique when training data is limited as is very often the case in practice. While cats and dogs are unlikely to produce tradeable signals, transfer learning could certainly help improve the accuracy of predictions on a relevant alternative dataset, such as the satellite images that we'll tackle next.

Classifying satellite images with transfer learning

Satellite images figure prominently among alternative data (see *Chapter 3*, *Alternative Data for Finance – Categories and Use Cases*). For instance, commodity traders may rely on satellite images to predict the supply of certain crops or resources by monitoring, activity on farms, at mining sites, or oil tanker traffic.

The EuroSat dataset

To illustrate working with this type of data, we load the EuroSat dataset included in the TensorFlow 2 datasets (Helber et al. 2019). The EuroSat dataset includes around 27,000 images in 64×64 format that represent 10 different types of land uses. *Figure 18.10* displays an example for each label:

Figure 18.10: Ten types of land use contained in the dataset

A time series of similar data could be used to track the relative sizes of cultivated, industrial, and residential areas or the status of specific crops to predict harvest quantities or quality, for example, for wine.

Fine-tuning a very deep CNN – DenseNet201

Huang et al. (2018) developed a new architecture dubbed **densely connected** based on the insight that CNNs can be deeper, more accurate, and more efficient to train if they contain shorter connections between layers close to the input and those close to the output.

One architecture, labeled **DenseNet201**, connects each layer to every other layer in a feedforward fashion. It uses the feature maps of all preceding layers as inputs, while each layer's own feature maps become inputs into all subsequent layers.

We download the DenseNet201 architecture from `tensorflow.keras.applications` and replace its final layers with the following dense layers interspersed with batch normalization to mitigate exploding or vanishing gradients in this very deep network with over 700 layers:

```
Layer (type)                 Output Shape              Param #
densenet201 (Model)          (None, 1920)              18321984
batch_normalization (BatchNo (None, 1920)              7680
dense (Dense)                (None, 2048)              3934208
batch_normalization_1 (Batch (None, 2048)              8192
dense_1 (Dense)              (None, 2048)              4196352
batch_normalization_2 (Batch (None, 2048)              8192
dense_2 (Dense)              (None, 2048)              4196352
batch_normalization_3 (Batch (None, 2048)              8192
dense_3 (Dense)              (None, 2048)              4196352
batch_normalization_4 (Batch (None, 2048)              8192
dense_4 (Dense)              (None, 10)                20490
Total params: 34,906,186
Trainable params: 34,656,906
Non-trainable params: 249,280
```

Model training and results evaluation

We use 10 percent of the training images for validation purposes and achieve the best out-of-sample classification accuracy of 97.96 percent after 10 epochs. This exceeds the performance cited in the original paper for the best-performing ResNet-50 architecture with a 90-10 split.

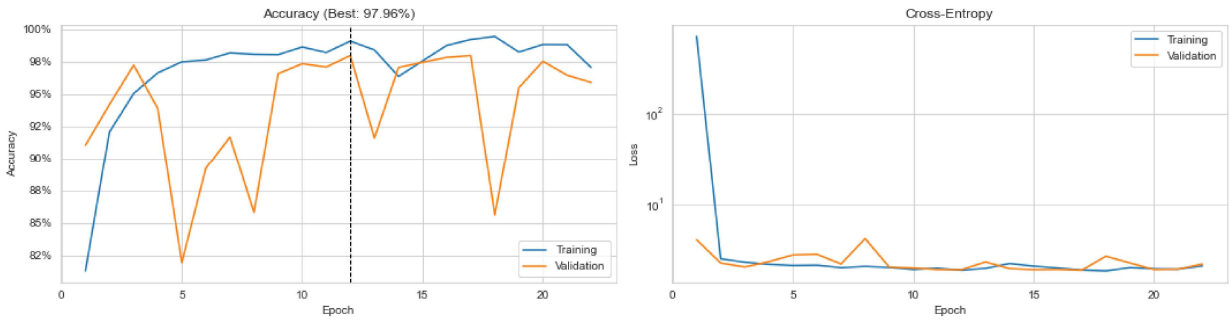


Figure 18.11: Cross-validation performance

There would likely be additional performance gains from augmenting the relatively small training set.

Object detection and segmentation

Image classification is a fundamental computer vision task that requires labeling an image based on certain objects it contains. Many practical applications, including investment and trading strategies, require additional information:

- The **object detection** task requires not only the identification but also the spatial location of all objects of interest, typically using bounding boxes. Several algorithms have been developed to overcome the inefficiency of brute-force sliding-window approaches, including region proposal methods (R-CNN; see for example Ren et al. 2015) and the **You Only Look Once** (**YOLO**) real-time object detection algorithm (Redmon 2016).
- The **object segmentation** task goes a step further and requires a class label and an outline of every object in the input image. This may be useful to count objects such as oil tankers, individuals, or cars in an image and evaluate a level of activity.
- **Semantic segmentation**, also called scene parsing, makes dense predictions to assign a class label to each pixel in the image. As a result, the image is divided into semantic regions and each pixel is assigned to its enclosing object or region.

Object detection requires the ability to distinguish between several classes of objects and to decide how many and which of these objects are present in an image.

Object detection in practice

A prominent example is Ian Goodfellow's identification of house numbers from Google's **Street View House Numbers** (**SVHN**) dataset (Goodfellow 2014). It requires the model to identify the following:

- How many of up to five digits make up the house number
- The correct digit for each component
- The proper order of the constituent digits

We will show how to preprocess the irregularly shaped source images, adapt the VGG16 architecture to produce multiple outputs, and train the final layer, before fine-tuning the pretrained weights to address the task.

Preprocessing the source images

The notebook `svhn_preprocessing.ipynb` contains code to produce a simplified, cropped dataset that uses bounding box information to create regularly shaped 32×32 images containing the digits; the original images are of arbitrary shape (Netzer 2011).

Figure 18.12: Cropped sample images of the SVHN dataset

The SVHN dataset contains house numbers with up to five digits and uses the class 10 if a digit is not present. However, since there are very few examples with five digits, we limit the images to those including up to four digits only.

Transfer learning with a custom final layer

The notebook `svhn_object_detection.ipynb` illustrates how to apply transfer learning to a deep CNN based on the VGG16 architecture, as outlined in the previous section. We will describe how to create new final layers that produce several outputs to meet the three SVHN task objectives, including one prediction of how many digits are present, and one for the value of each digit in the order they appear.

The best-performing architecture on the original dataset has eight convolutional layers and two final fully connected layers. We will use **transfer learning**, departing from the VGG16 architecture. As before, we import the VGG16 network pretrained on ImageNet weights, remove the layers after the convolutional blocks, freeze the weights, and create new dense and predictive layers as follows using the Functional API:

```
vgg16 = VGG16(input_shape=IMG_SHAPE, include_top=False, weights='imagenet')
vgg16.trainable = False
x = vgg16.output
x = Flatten()(x)
x = BatchNormalization()(x)
x = Dense(256)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = Dense(128)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
n_digits = Dense(SEQ_LENGTH, activation='softmax', name='n_digits')(x)
digit1 = Dense(N_CLASSES-1, activation='softmax', name='d1')(x)
digit2 = Dense(N_CLASSES, activation='softmax', name='d2')(x)
digit3 = Dense(N_CLASSES, activation='softmax', name='d3')(x)
digit4 = Dense(N_CLASSES, activation='softmax', name='d4')(x)
predictions = Concatenate()([n_digits, digit1, digit2, digit3, digit4])
```

The prediction layer combines the four-class output for the number of digits `n_digits` with four outputs that predict which digit is present at that position.

Creating a custom loss function and evaluation metrics

The custom output requires us to define a loss function that captures how well the model is meeting its objective. We would also like to measure accuracy in a way that reflects predictive accuracy tailored to the specific labels.

For the custom loss, we average the cross-entropy over the five categorical outputs, namely the number of digits and their respective values:

```
def weighted_entropy(y_true, y_pred):
    cce = tf.keras.losses.SparseCategoricalCrossentropy()
    n_digits = y_pred[:, :SEQ_LENGTH]
    digits = {}
    for digit, (start, end) in digit_pos.items():
        digits[digit] = y_pred[:, start:end]
    return (cce(y_true[:, 0], n_digits) +
            cce(y_true[:, 1], digits[1]) +
            cce(y_true[:, 2], digits[2]) +
            cce(y_true[:, 3], digits[3]) +
            cce(y_true[:, 4], digits[4])) / 5
```

To measure predictive accuracy, we compare the five predictions with the corresponding label values and average the share of correct matches over the batch of samples:

```
def weighted_accuracy(y_true, y_pred):
    n_digits_pred = K.argmax(y_pred[:, :SEQ_LENGTH], axis=1)

    digit_preds = {}
    for digit, (start, end) in digit_pos.items():
        digit_preds[digit] = K.argmax(y_pred[:, start:end], axis=1)
    preds = tf.dtypes.cast(tf.stack((n_digits_pred,
                                     digit_preds[1],
                                     digit_preds[2],
                                     digit_preds[3],
                                     digit_preds[4]), axis=1), tf.float32)

    return K.mean(K.sum(tf.dtypes.cast(K.equal(y_true, preds), tf.int64),
axis=1) / 5)
```

Finally, we integrate the base and final layers and compile the model with the custom loss and accuracy metric as follows:

```
model = Model(inputs=vgg16.input, outputs=predictions)
model.compile(optimizer='adam',
              loss=weighted_entropy,
              metrics=[weighted_accuracy])
```

Fine-tuning the VGG16 weights and final layer

We train the new final layers for 14 periods and continue fine-tuning all VGG16 weights, as in the previous section, for another 23 epochs (using early stopping in both cases).

The following charts show the training and validation accuracy and the loss over the entire training period. As we unfreeze the VGG16 weights after the initial training period, the accuracy drops and then improves, achieving a validation performance of 94.52 percent:

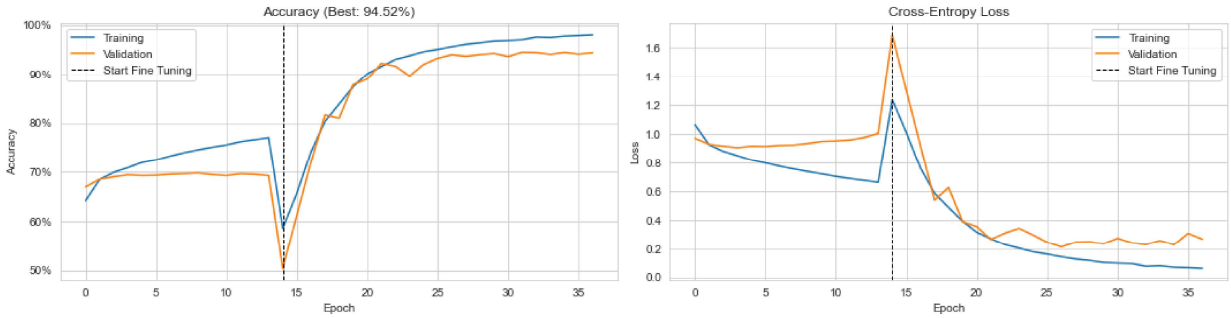


Figure 18.13: Cross-validation performance

See the notebook for additional implementation details and an evaluation of the results.

Lessons learned

We can achieve decent levels of accuracy using only the small training set. However, state-of-the-art performance achieves an error rate of only 1.02 percent (`https://benchmarks.ai/svhn`). To get closer, the most important step is to increase the amount of training data.

There are two easy ways to accomplish this: we can include the larger number of samples included in the **extra** dataset, and we can use image augmentation (see the *AlexNet: reigniting deep learning research* section). The currently best-performing approach relies heavily on augmentation learned from data (Cubuk 2019).

CNNs for time-series data – predicting returns

CNNs were originally developed to process image data and have achieved superhuman performance on various computer vision tasks. As discussed in the first section, time-series data has a grid-like structure similar to that of images, and CNNs have been successfully applied to one-, two- and three-dimensional representations of temporal data.

The application of CNNs to time series will most likely bear fruit if the data meets the model's key assumption that local patterns or relationships help predict the outcome. In the time-series context, local patterns could be autocorrelation or similar non-linear relationships at relevant intervals. Along the second and third dimensions, local patterns imply systematic relationships among different components of a multivariate series or among these series for different tickers. Since locality matters, it is important that the data is organized accordingly, in contrast to feed-forward networks where shuffling the elements of any dimension does not negatively affect the learning process.

In this section, we provide a relatively simple example using a one-dimensional convolution to model an autoregressive process (see *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*) that predicts future returns based on lagged returns. Then we replicate a recent research paper that achieved good results by formatting multivariate time-series data like images to predict returns. We will also develop and test a trading strategy based on the signals contained in the predictions.

An autoregressive CNN with 1D convolutions

We will introduce the time series use case for CNN using a univariate autoregressive asset return model. More specifically, the model receives the most recent 12 months of returns and uses a single layer of one-dimensional convolutions to predict the subsequent month.

The requisite steps are as follows:

1. Creating the rolling 12 months of lagged returns and corresponding outcomes
2. Defining the model architecture
3. Training the model and evaluating the results

In the following sections, we'll describe each step in turn; the notebook `time_series_prediction` contains the code samples for this section.

Preprocessing the data

First, we'll select the adjusted close price for all Quandl Wiki stocks since 2000 as follows:

```
prices = (pd.read_hdf('../data/assets.h5', 'quandl/wiki/prices')
          .adj_close
          .unstack().loc['2000':])
prices.info()
DatetimeIndex: 2896 entries, 2007-01-01 to 2018-03-27
Columns: 3199 entries, A to ZUMZ
```

Next, we resample the price data to month-end frequency, compute returns, and set monthly returns over 100 percent to missing as they likely represent data errors. Then we drop tickers with missing observations, retaining 1,511 stocks with 215 observations each:

```
returns = (prices
           .resample('M')
           .last()
           .pct_change()
           .dropna(how='all')
           .loc['2000': '2017']
           .dropna(axis=1)
           .sort_index(ascending=False))

# remove outliers likely representing data errors
returns = returns.where(returns<1).dropna(axis=1)
returns.info()
DatetimeIndex: 215 entries, 2017-12-31 to 2000-02-29
Columns: 1511 entries, A to ZQK
```

To create the rolling series of 12 lagged monthly returns with their corresponding outcome, we iterate over rolling 13-month slices and append the transpose of each slice to a list after assigning the outcome date to the index. After completing the loop, we concatenate the DataFrames in the list as follows:

```
n = len(returns)
nlags = 12
lags = list(range(1, nlags + 1))
cnn_data = []
for i in range(n-nlags-1):
    df = returns.iloc[i:i+nlags+1]        # select outcome and lags
    date = df.index.max()                 # use outcome date
    cnn_data.append(df.reset_index(drop=True)  # append transposed series
                    .transpose()
                    .assign(date=date)
                    .set_index('date', append=True)
                    .sort_index(1, ascending=True))
```

```
cnn_data = (pd.concat(cnn_data)
            .rename(columns={0: 'label'})
            .sort_index())
```

We end up with over 305,000 pairs of outcomes and lagged returns for the 2001-2017 period:

```
cnn_data.info(null_counts=True)
MultiIndex: 305222 entries, ('A', Timestamp('2001-03-31 00:00:00')) to
                            ('ZQK', Timestamp('2017-12-31 00:00:00'))
Data columns (total 13 columns):
...
```

When we compute the information coefficient for each lagged return and the outcome, we find that only lag 5 is not statistically significant:

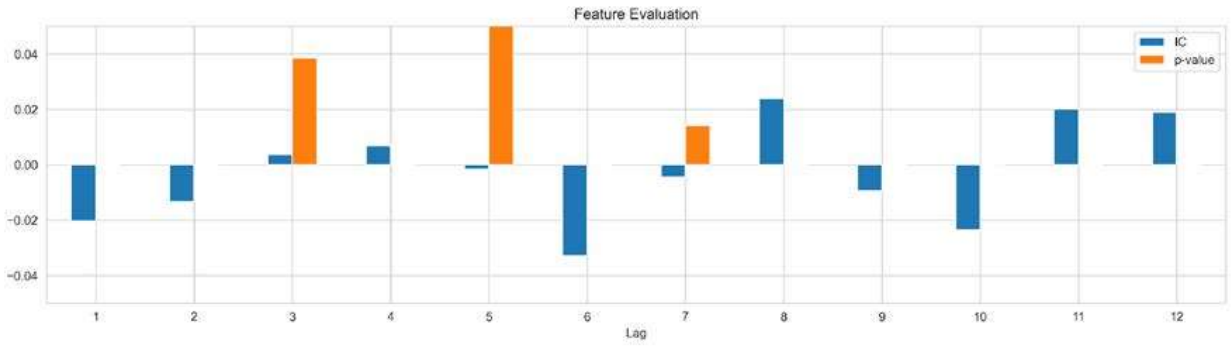


Figure 18.14: Information coefficient with respect to forward return by lag

Defining the model architecture

Now we'll define the model architecture using TensorFlow's Keras interface. We combine a one-dimensional convolutional layer with max pooling and batch normalization to produce a real-valued scalar output:

```
model = Sequential([Conv1D(filters=32,
                           kernel_size=4,
                           activation='relu',
                           padding='causal',
                           input_shape=(12, 1),
                           use_bias=True,
                           kernel_regularizer=regularizers.l1_l2(l1=1e-5,
                                                                 l2=1e-5)),
                    MaxPooling1D(pool_size=4),
                    Flatten(),
                    BatchNormalization(),
                    Dense(1, activation='linear')])
```

The one-dimensional convolution computes the sliding dot product of a (regularized) vector of length 4 with each input sequence of length 12, using causal padding to maintain the temporal order (see the *How to scan the input: strides and padding* section). The resulting 32 feature maps have the same length, 12, as the input that max pooling in groups of size 4 reduces to 32 vectors of length 3.

The model outputs the weighted average plus the bias of the flattened and normalized single vector of length 96, and has 449 trainable parameters:

```
Layer (type)                 Output Shape              Param #
conv1d (Conv1D)              (None, 12, 32)            160
max_pooling1d (MaxPooling1D) (None, 3, 32)             0
flatten (Flatten)            (None, 96)                0
batch_normalization (BatchNo (None, 96)                384
dense (Dense)                (None, 1)                 97
Total params: 641
Trainable params: 449
Non-trainable params: 192
```

The notebook wraps the model generation and subsequent compilation into a `get_model()` function that parametrizes the model configuration to facilitate experimentation.

Model training and performance evaluation

We train the model on five years of data for each ticker to predict the first month after this period and repeat this procedure 36 times using the `MultipleTimeSeriesCV` we developed in *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*. See the notebook for the training loop that follows the pattern demonstrated in the previous chapter.

We use early stopping after five epochs to simplify the exposition, resulting in a positive bias so that the results have only illustrative character. Training length varies from 1 to 27 epochs, with a median of 5 epochs, which demonstrates that the model can often only learn very limited amounts of systematic information from the past returns. Thus cherry-picking the results yields a cumulative average information coefficient of around 4, as shown in *Figure 18.15*:

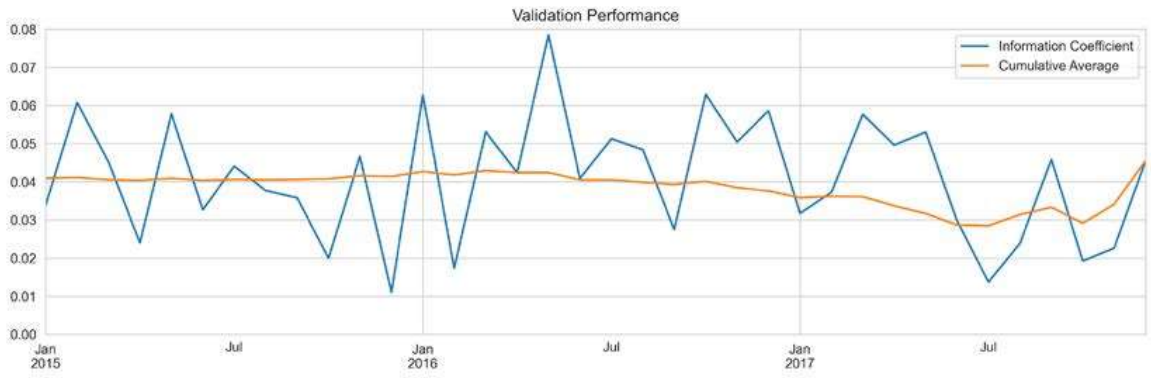


Figure 18.15: (Biased) out-of-sample information coefficients for best epochs

We'll now proceed to a more complex example of using CNNs for multiple time-series data.

CNN-TA – clustering time series in 2D format

To exploit the grid-like structure of time-series data, we can use CNN architectures for univariate and multivariate time series. In the latter case, we consider different time series as channels, similar to the different color signals.

An alternative approach converts a time series of alpha factors into a two-dimensional format to leverage the ability of CNNs to detect local patterns. Sezer and Ozbayoglu (2018) propose **CNN-TA**, which computes 15 technical indicators for different intervals and uses hierarchical clustering (see *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*) to locate indicators that behave similarly close to each other in a two-dimensional grid.

The authors train a CNN similar to the CIFAR-10 example we used earlier to predict whether to buy, hold, or sell an asset on a given day. They compare the CNN performance to "buy-and-hold" and other models and find that it outperforms all alternatives using daily price series for Dow 30 stocks and the nine most-traded ETFs over the 2007-2017 time period.

In this section, we experiment with this approach using daily US equity price data and demonstrate how to compute and convert a similar set of indicators into image format. Then we train a CNN to predict daily returns and evaluate a simple long-short strategy based on the resulting signals.

Creating technical indicators at different intervals

We first select a universe of the 500 most-traded US stocks from the Quandl Wiki dataset by dollar volume for rolling five-year periods for 2007-2017. See the notebook `engineer_cnn_features.ipynb` for the code examples in this section and some additional implementation details.

Our features consist of 15 technical indicators and risk factors that we compute for 15 different intervals and then arrange them in a 15×15 grid. The following table lists some of the technical indicators; in addition, we follow the authors in using the following metrics (see the *Appendix* for additional information):

- **Weighted and exponential moving averages** (**WMA** and **EMA**) of the close price
- **Rate of change** (**ROC**) of the close price
- **Chande Momentum Oscillator** (**CMO**)
- **Chaikin A/D Oscillators** (**ADOSC**)
- **Average Directional Movement Index** (**ADX**)

Indicator	Name	Formula
Relative Strength Index (RSI)	Oscillates in [0, 100] range; below 30: oversold, over 70: overbought	See Chapter 4
Williams %R	Momentum-based in [-100, 0] range, below -80: oversold, above -20: overbought	$R = \frac{\max(\mathbf{high}) - \mathbf{close}}{\max(\mathbf{high}) - \min(\mathbf{low})}$
Bollinger Bands	20-day moving average plus/minus daily standard deviation; prices above/below these bands indicate overbought/sold	See chapter 4.
Normalized Average True Range (NATR)	Avg. true range: max of current high-low, current high-prev.close or absolute of prev. close - current low, averaged over t days.	$\mathbf{NATR} = \frac{\mathbf{ATR}(t)}{\mathbf{Close}}$
Percentage Price Oscillator (PPO)	Momentum: compares two exponential moving averages (EMA) in percentage terms	$\mathbf{PPO} = \frac{\mathbf{EMA}_{12} - \mathbf{EMA}_{26}}{\mathbf{EMA}_{26}}$
Commodity Channel Index (CCI)	Momentum-based: difference between current and simple moving average (SMA) of the historical average price, normalized by their mean difference	$p^{\mathbf{hist}} = \sum_{i=1}^{P} (\mathbf{high} + \mathbf{low} + \mathbf{close})/3$ $\mathbf{CCI} = \frac{p^{\mathbf{hist}} - \mathbf{SMA}\left(p^{\mathbf{hist}}\right)}{0.15 \times \sum_{i=1}^{P} \lvert p^{\mathbf{hist}} - \mathbf{SMA}\left(p^{\mathbf{hist}}\right)\rvert / P}$

Figure 8.16: Technical indicators

For each indicator, we vary the time period from 6 to 20 to obtain 15 distinct measurements. For example, the following code example computes the **relative strength index** (**RSI**):

```
T = list(range(6, 21))
for t in T:
    universe[f'{t:02}_RSI'] = universe.groupby(level='symbol').close.
apply(RSI, timeperiod=t)
```

For the **Normalized Average True Range** (**NATR**) that requires several inputs, the computation works as follows:

```
for t in T:
    universe[f'{t:02}_NATR'] = universe.groupby(
                        level='symbol', group_keys=False).apply(
                        lambda x: NATR(x.high, x.low, x.close, timeperiod=t))
```

See the TA-Lib documentation for further details.

Computing rolling factor betas for different horizons

We also use **five Fama-French risk factors** (**Fama** and French, 2015; see *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*). They reflect the sensitivity of a stock's returns to factors consistently demonstrated to impact equity returns. We capture these factors by computing the coefficients of a rolling OLS regression of a stock's daily returns on the returns of portfolios designed to reflect the underlying drivers:

- **Equity risk premium**: Value-weighted returns of US stocks minus the 1-month US Treasury bill rate
- **Size** (**SMB**): Returns of stocks categorized as **Small** (by market cap) **Minus** those of **Big equities**

- **Value** (**HML**): Returns of stocks with **High** book-to-market value **Minus** those with a **Low value**
- **Investment** (**CMA**): Returns differences for companies with **Conservative** investment expenditures **Minus** those with **Aggressive spending**
- **Profitability** (**RMW**): Similarly, return differences for stocks with **Robust** profitability **Minus** that with a **Weak** metric.

We source the data from Kenneth French's data library using `pandas_datareader` (see *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*):

```
import pandas_datareader.data as web
factor_data = (web.DataReader('F-F_Research_Data_5_Factors_2x3_daily',
                              'famafrench', start=START)[0])
```

Next, we apply statsmodels' `RollingOLS()` to run regressions over windowed periods of different lengths, ranging from 15 to 90 days. We set the `params_only` parameter on the `.fit()` method to speed up computation and capture the coefficients using the `.params` attribute of the fitted `factor_model`:

```
factors = [Mkt-RF, 'SMB', 'HML', 'RMW', 'CMA']
windows = list(range(15, 90, 5))
for window in windows:
    betas = []
    for symbol, data in universe.groupby(level='symbol'):
        model_data = data[[ret]].merge(factor_data, on='date').dropna()
        model_data[ret] -= model_data.RF
        rolling_ols = RollingOLS(endog=model_data[ret],
                                 exog=sm.add_constant(model_data[factors]),
                                                      window=window)
        factor_model = rolling_ols.fit(params_only=True).params.drop('const',
                                                                     axis=1)
        result = factor_model.assign(symbol=symbol).set_index('symbol',
                                                              append=True)
        betas.append(result)
    betas = pd.concat(betas).rename(columns=lambda x: f'{window:02}_{x}')
    universe = universe.join(betas)
```

Features selecting based on mutual information

The next step is to select the 15 most relevant features from the 20 candidates to fill the 15×15 input grid. The code examples for the following steps are in the notebook `convert_cnn_features_to_image_format`.

To this end, we estimate the mutual information for each indicator and the 15 intervals with respect to our target, the one-day forward returns. As discussed in *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*, scikit-learn provides the `mutual_info_regression()` function that makes this straightforward, albeit time-consuming and memory-intensive. To accelerate the process, we randomly sample 100,000 observations:

```
df = features.join(targets[target]).dropna().sample(n=100000)
X = df.drop(target, axis=1)
y = df[target]
mi[t] = pd.Series(mutual_info_regression(X=X, y=y), index=X.columns)
```

The left panel in *Figure 18.16* shows the mutual information, averaged across the 15 intervals for each indicator. NATR, PPO, and Bollinger Bands are most important from this metric's perspective:

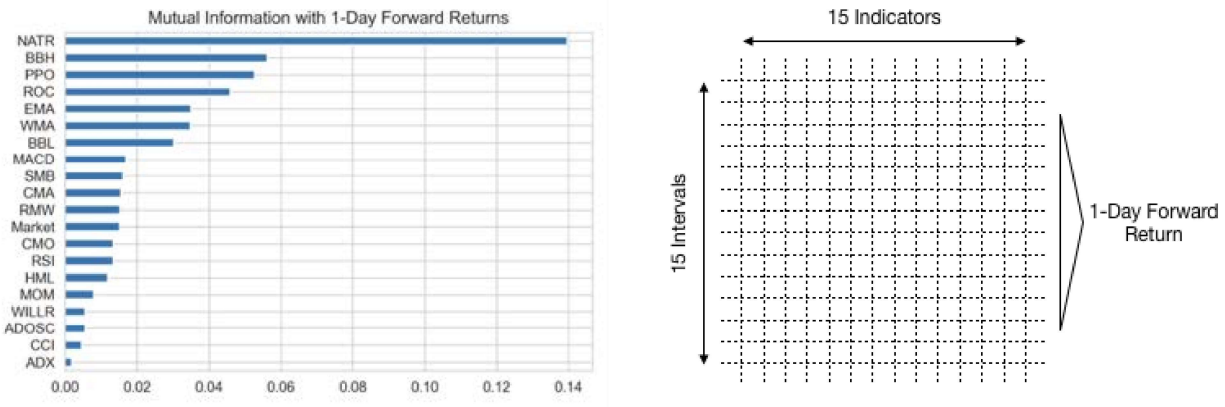


Figure 18.17: Mutual information and two-dimensional grid layout for time series

Hierarchical feature clustering

The right panel in *Figure 18.16* sketches the 15 X 15 two-dimensional feature grid that we will feed into our CNN. As discussed in the first section of this chapter, CNNs rely on the locality of relevant patterns that is typically found in images where nearby pixels are closely related and changes from one pixel to the next are often gradual.

To organize our indicators in a similar fashion, we will follow Sezer and Ozbayoglu's approach of applying hierarchical clustering. The goal is to identify features that behave similarly and order the columns and the rows of the grid accordingly.

We can build on SciPy's `pairwise_distance()`, `linkage()`, and `dendrogram()` functions that we introduced in *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning* alongside other forms of clustering. We create a helper function that standardizes the input column-wise to avoid distorting distances among features due to differences in scale, and use the Ward criterion that merges clusters to minimize variance. The function returns the order of the leaf nodes in the dendrogram that in turn displays the successive formation of larger clusters:

```
def cluster_features(data, labels, ax, title):
    data = StandardScaler().fit_transform(data)
    pairwise_distance = pdist(data)
    Z = linkage(data, 'ward')
    dend = dendrogram(Z,
                      labels=labels,
                      orientation='top',
                      leaf_rotation=0.,
                      leaf_font_size=8.,
                      ax=ax)
    return dend['ivl']
```

To obtain the optimized order of technical indicators in the columns and the different intervals in the rows, we use NumPy's `.reshape()` method to ensure that the dimension we would like to cluster appears in the columns of the two-dimensional array we pass to `cluster_features()`:

```
labels = sorted(best_features)
col_order = cluster_features(features.dropna().values.reshape(-1, 15).T,
                             labels)

labels = list(range(1, 16))
row_order = cluster_features(
    features.dropna().values.reshape(-1, 15, 15).transpose((0, 2, 1)).
reshape(-1, 15).T, labels)
```

Figure 18.18 shows the dendrograms for both the row and column features:

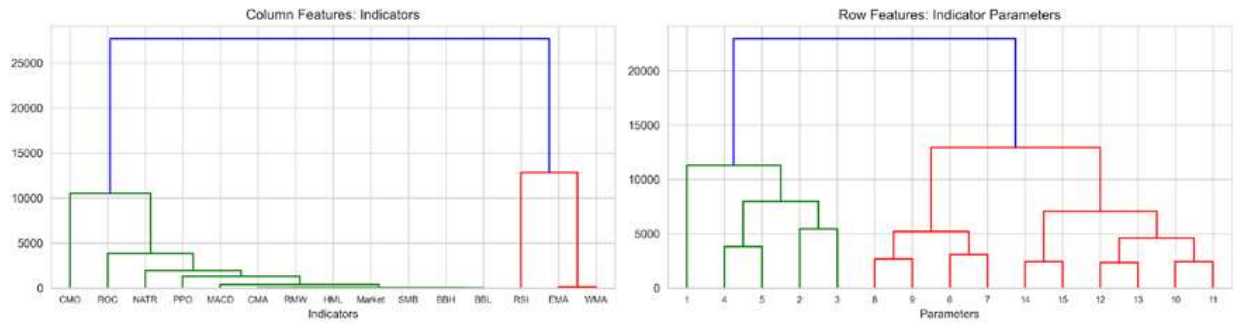


Figure 18.18: Dendrograms for row and column features

We reorder the features accordingly and store the result as inputs for the CNN that we will create in the next step.

Creating and training a convolutional neural network

Now we are ready to design, train, and evaluate a CNN following the steps outlined in the previous section. The notebook `cnn_for_trading.ipynb` contains the relevant code examples.

We again closely follow the authors in creating a CNN with 2 convolutional layers with kernel size 3 and 16 and 32 filters, respectively, followed by a max pooling layer of size 2. We flatten the output of the last stack of filters and connect the resulting 1,568 outputs to a dense layer of size 32, applying 25 and 50 percent dropout probability to the incoming and outcoming connections to mitigate overfitting. The following table summarizes the CNN structure that contains 55,041 trainable parameters:

```
Layer (type)                 Output Shape              Param #
CONV1 (Conv2D)               (None, 15, 15, 16)        160
CONV2 (Conv2D)               (None, 15, 15, 32)        4640
POOL1 (MaxPooling2D)         (None, 7, 7, 32)          0
DROP1 (Dropout)              (None, 7, 7, 32)          0
FLAT1 (Flatten)              (None, 1568)              0
FC1 (Dense)                  (None, 32)                50208
DROP2 (Dropout)              (None, 32)                0
FC2 (Dense)                  (None, 1)                 33
Total params: 55,041
Trainable params: 55,041
Non-trainable params: 0
```

We cross-validate the model with the `MutipleTimeSeriesCV` train and validation set index generator introduced in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. We provide 5 years of trading days during the training period in batches of 64 random samples and validate using the subsequent 3 months, covering the years 2014-2017.

We scale the features to the range [-1, 1] and again use NumPy's `.reshape()` method to create the requisite $N \times 15 \times 15 \times 1$ format:

```
def get_train_valid_data(X, y, train_idx, test_idx):
    x_train, y_train = X.iloc[train_idx, :], y.iloc[train_idx]
    x_val, y_val = X.iloc[test_idx, :], y.iloc[test_idx]
    scaler = MinMaxScaler(feature_range=(-1, 1))
    x_train = scaler.fit_transform(x_train)
    x_val = scaler.transform(x_val)
    return (x_train.reshape(-1, size, size, 1), y_train,
            x_val.reshape(-1, size, size, 1), y_val)
```

Training and validation follow the process laid out in *Chapter 17, Deep Learning for Trading*, relying on checkpointing to store weights after each epoch and generate predictions for the best-performing iterations without the need for costly retraining.

To evaluate the model's predictive accuracy, we compute the daily **information coefficient** (**IC**) for the validation set like so:

```
checkpoint_path = Path('models', 'cnn_ts')
for fold, (train_idx, test_idx) in enumerate(cv.split(features)):
    X_train, y_train, X_val, y_val = get_train_valid_data(features, target,
train_idx, test_idx)
```

```
    preds = y_val.to_frame('actual')
    r = pd.DataFrame(index=y_val.index.unique(level='date')).sort_index()
    model = make_model(filter1=16, act1='relu', filter2=32,
                       act2='relu', do1=.25, do2=.5, dense=32)
    for epoch in range(n_epochs):
        model.fit(X_train, y_train,
                  batch_size=batch_size,
                  validation_data=(X_val, y_val),
                  epochs=1, verbose=0, shuffle=True)
        model.save_weights(
            (checkpoint_path / f'ckpt_{fold}_{epoch}').as_posix())
        preds[epoch] = model.predict(X_val).squeeze()
        r[epoch] = preds.groupby(level='date').apply(
            lambda x: spearmanr(x.actual, x[epoch])[0]).to_frame(epoch)
```

We train the model for up to 10 epochs using **stochastic gradient descent** with **Nesterov** momentum (see *Chapter 17*, *Deep Learning for Trading*) and find that the best performing epochs, 8 and 9, achieve a (low) daily average IC of around 0.009.

Assembling the best models to generate tradeable signals

To reduce the variance of the test-period forecasts, we generate and average the predictions for the 3 models that perform best during cross-validation, which here correspond to training for 4, 8, and 9 epochs. As in the previous time-series example, the relatively short training period underscores that the amount of signals in financial time series is low compared to the systematic information contained in, for example, image data.

The `generate_predictions()` function reloads the model weights and returns the forecasts for the target period:

```
def generate_predictions(epoch):
    predictions = []
    for fold, (train_idx, test_idx) in enumerate(cv.split(features)):
        X_train, y_train, X_val, y_val = get_train_valid_data(
            features, target, train_idx, test_idx)
        preds = y_val.to_frame('actual')
        model = make_model(filter1=16, act1='relu', filter2=32,
                           act2='relu', do1=.25, do2=.5, dense=32)
        status = model.load_weights(
            (checkpoint_path / f'ckpt_{fold}_{epoch}').as_posix())
        status.expect_partial()
        predictions.append(pd.Series(model.predict(X_val).squeeze(),
                                     index=y_val.index))
    return pd.concat(predictions)

preds = {}
for i, epoch in enumerate(ic.drop('fold', axis=1).mean().nlargest(3).index):
    preds[i] = generate_predictions(epoch)
```

We store the predictions and proceed to backtest a trading strategy based on these daily return forecasts.

Backtesting a long-short trading strategy

To get a sense of the signal quality, we compute the spread between equally weighted portfolios invested in stocks selected according to the signal quintiles using Alphalens (see *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*).

Figure 18.19 shows that for a one-day investment horizon, this naive strategy would have earned a bit over four basis points per day during the 2013-2017 period:

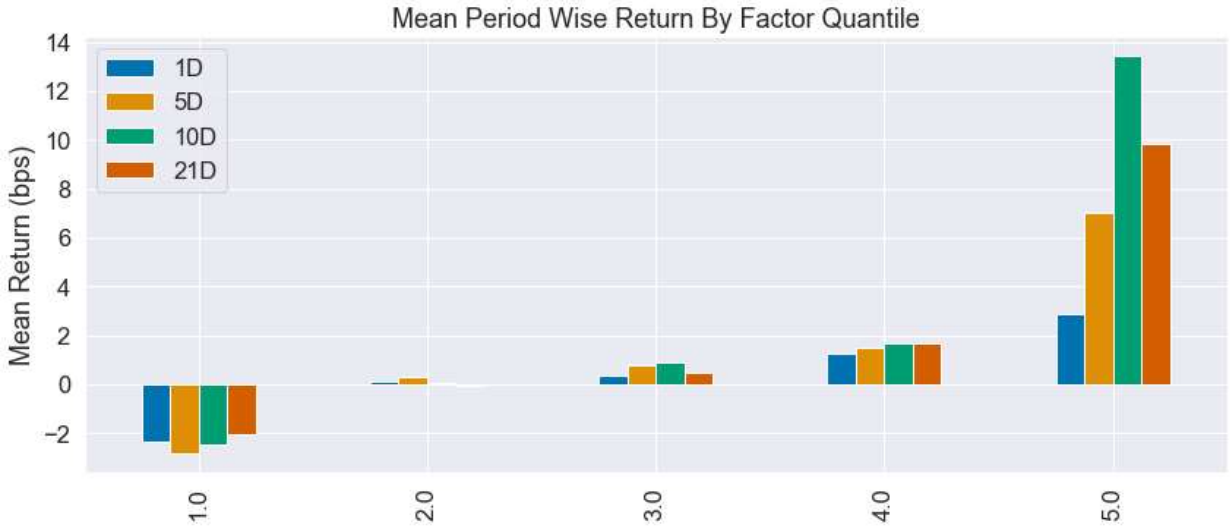


Figure 18.19: Alphalens signal quality evaluation

We translate this slightly encouraging result into a simple strategy that enters long (short) positions for the 25 stocks with the highest (lowest) return forecasts, trading on a daily basis. *Figure 18.20* shows that this strategy is competitive with the S&P 500 benchmark over much of the backtesting period (left panel), resulting in a 35.6 percent cumulative return and a Sharpe ratio of 0.53 (before transaction costs; right panel)

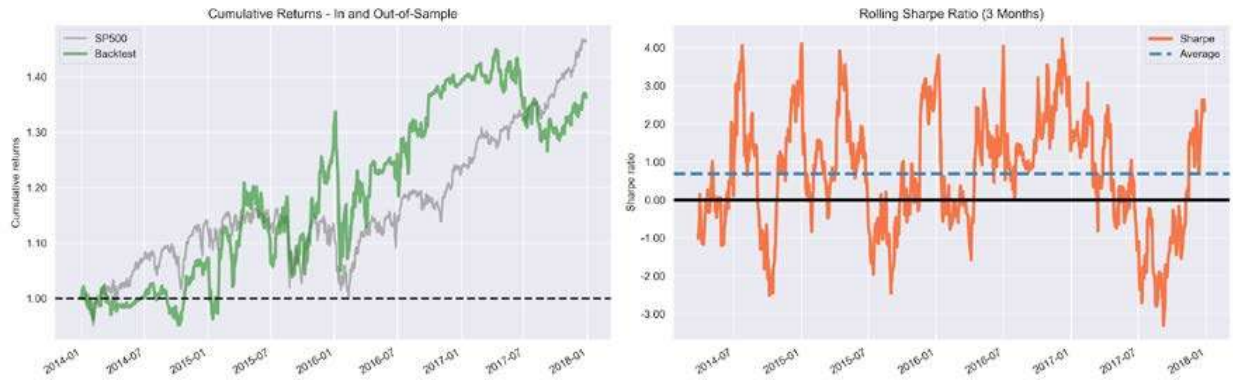


Figure 18.20: Backtest performance in- and out-of-sample

Summary and lessons learned

It appears that the CNN is able to extract meaningful information from the time series of alpha factors converted into a two-dimensional grid. Experimentation with different architectures and training parameters shows that the result is not very robust and slight modifications can yield significantly worse performance.

Tuning attempts also surface the notorious difficulties in successfully training a deep NN, especially when the signal-to-noise ratio is low: too complex a network or the wrong optimizer can lead the CNN to a local optimum where it always predicts a constant value.

The most important step to improve the results and obtain a performance closer to that achieved by the authors (using different outcomes) would be to revisit the features. There are many alternatives to different intervals of a limited set of technical indicators. Any appropriate number of time-series features could be arranged in a rectangular $n \times m$ format and benefit from the CNN's ability to learn local patterns. The choice of n indicators and m intervals just makes it easier to organize the rows and the columns of the two-dimensional grid. Give it a shot!

Furthermore, the authors take a classification approach to the algorithmically labeled buy, hold, and sell outcomes (see the paper for an outline of the computation), whereas our experiment applied regression to the daily returns. The Alphalens chart in *Figure 18.18* suggests that longer holding periods (especially 10 days) might work better, so there is also scope for adjusting the strategy accordingly or switching to a classification approach.

Summary

In this chapter, we introduced CNNs, a specialized NN architecture that has taken cues from our (limited) understanding of human vision and performs particularly well on grid-like data. We covered the central operation of convolution or cross-correlation that drives the discovery of filters that in turn detect features useful to solve the task at hand.

We reviewed several state-of-the-art architectures that are good starting points, especially because transfer learning enables us to reuse pretrained weights and reduce the otherwise rather computationally and data-intensive training effort. We also saw that Keras makes it relatively straightforward to implement and train a diverse set of deep CNN architectures.

In the next chapter, we turn our attention to recurrent neural networks that are designed specifically for sequential data, such as time-series data, which is central to investment and trading.

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 nonlinear transformations carried out by one or more neural network layers:

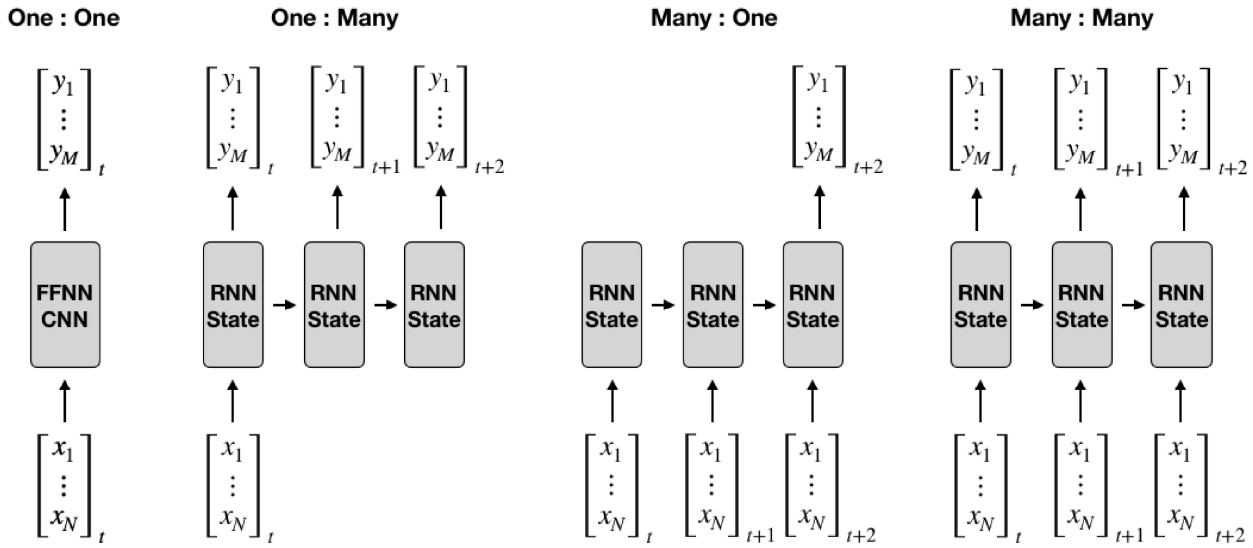


Figure 19.1: Various types of sequence-to-sequence models

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:

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

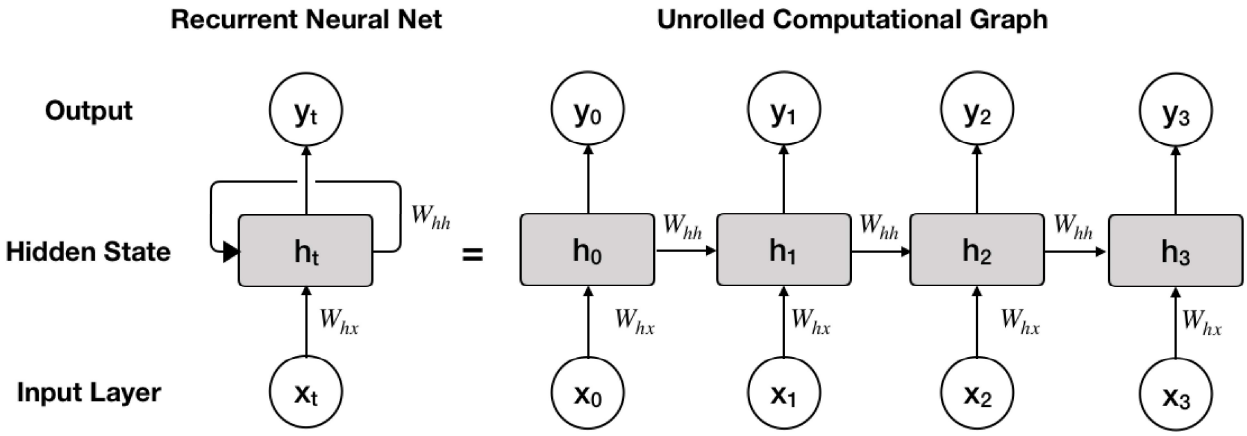


Figure 19.2: Recurrent and unrolled view of the computational graph of an RNN with a single hidden unit

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:

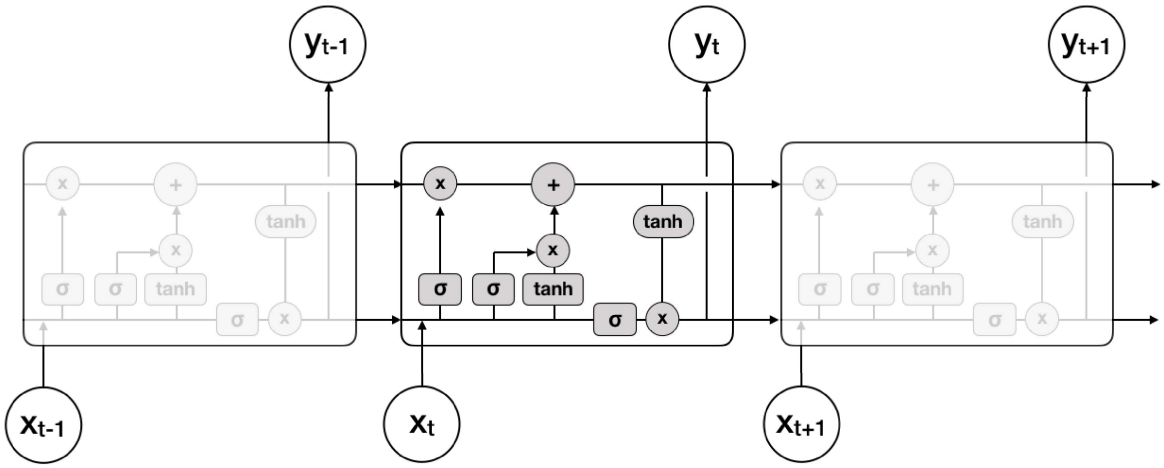
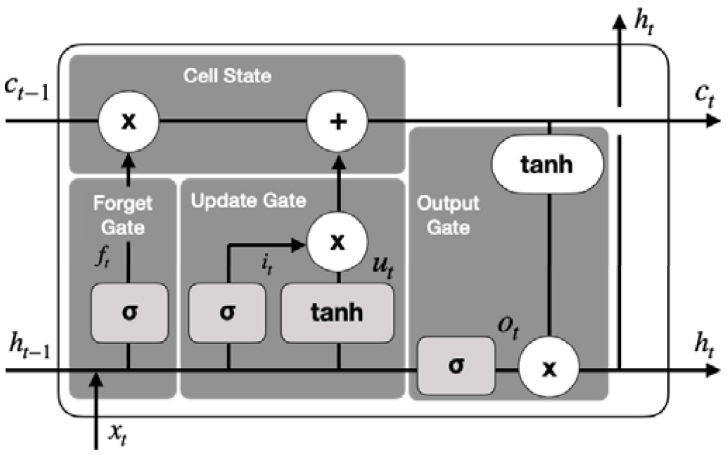


Figure 19.3: Information flow through an unrolled LSTM cell

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 element-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:

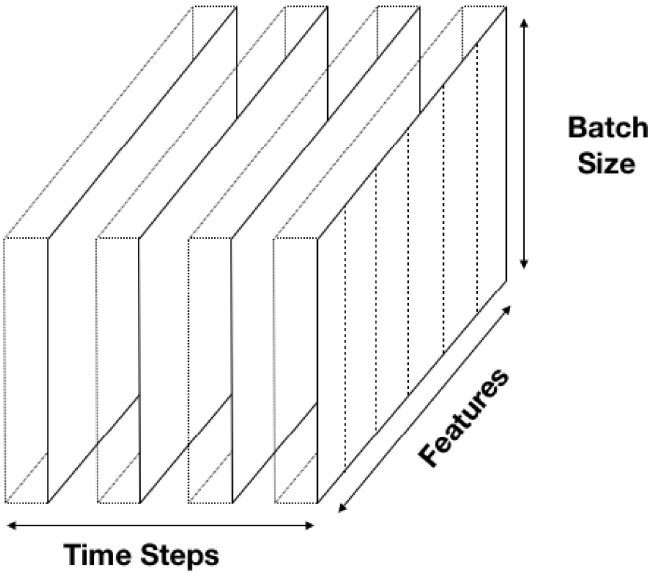


Figure 19.5: The three dimensions of an RNN input tensor

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)
```

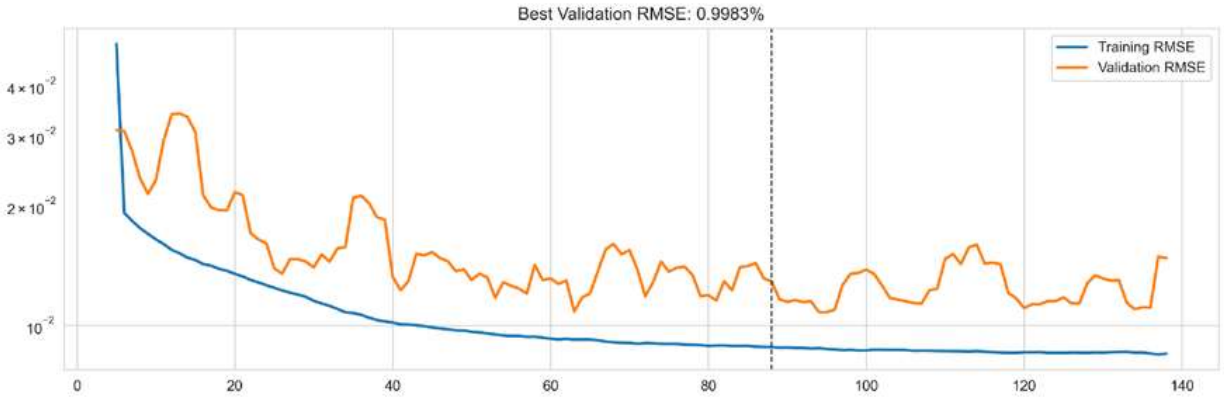


Figure 19.7: Cross-validation performance

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:

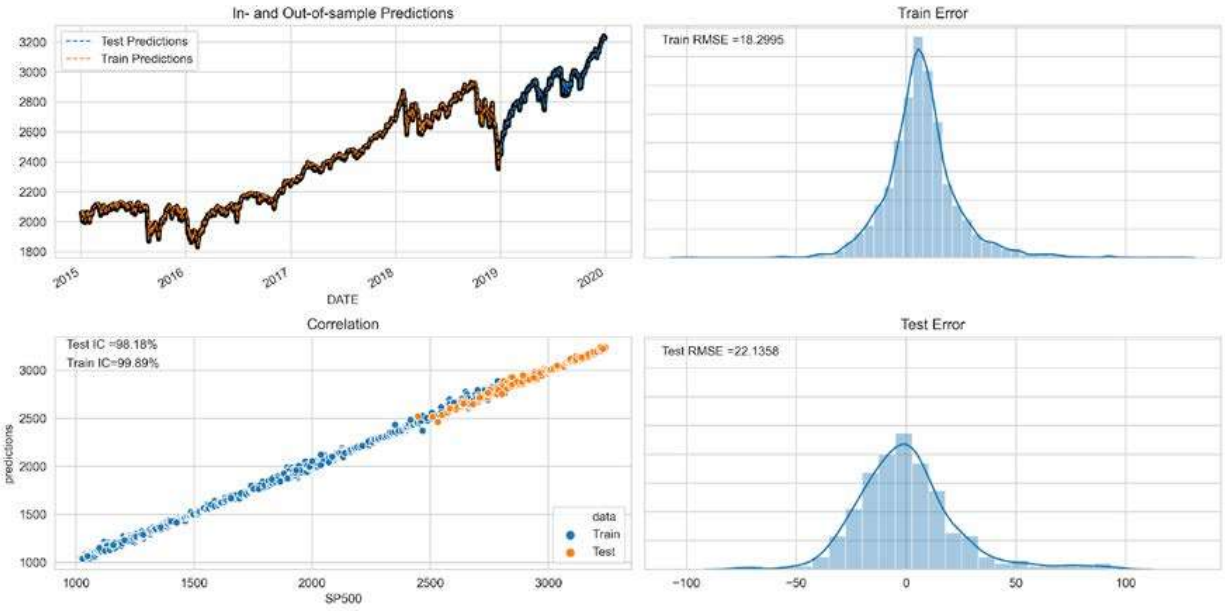


Figure 19.8: RNN performance on S&P 500 predictions

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:

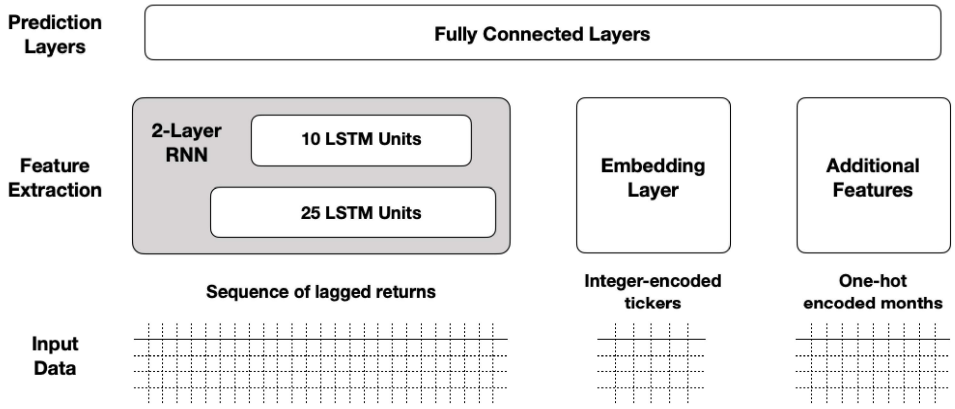


Figure 19.9: Stacked LSTM architecture with additional features

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:

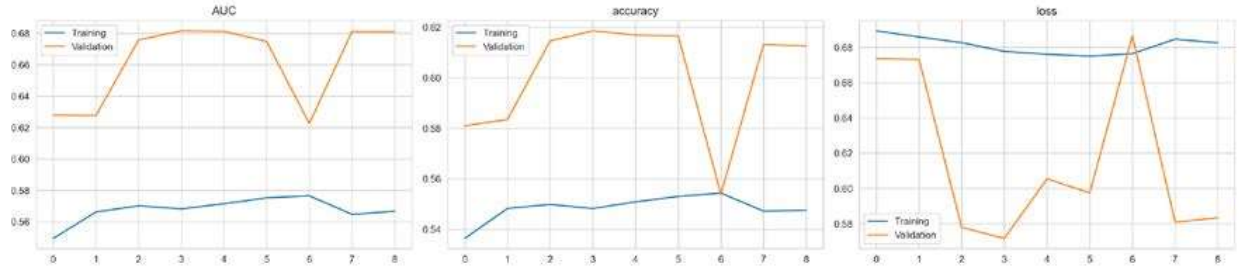


Figure 19.10: Stacked LSTM classification—cross-validation performance

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

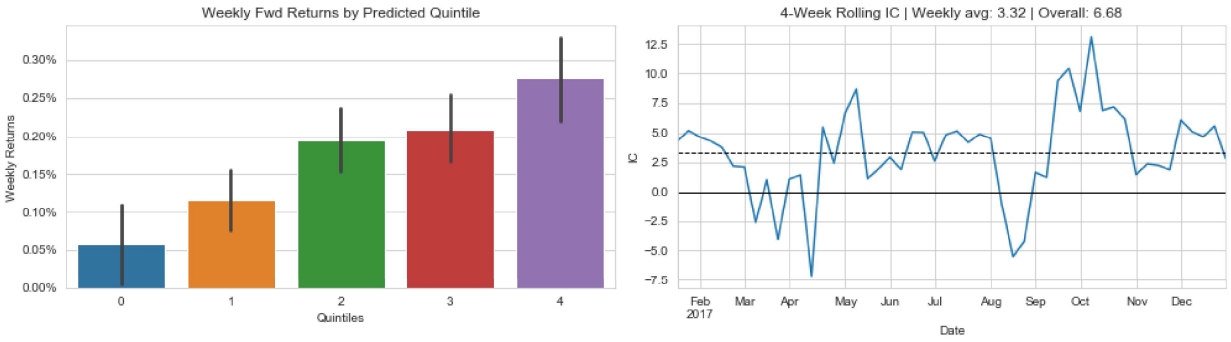


Figure 19.11: Stacked LSTM regression—out-of-sample performance

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:

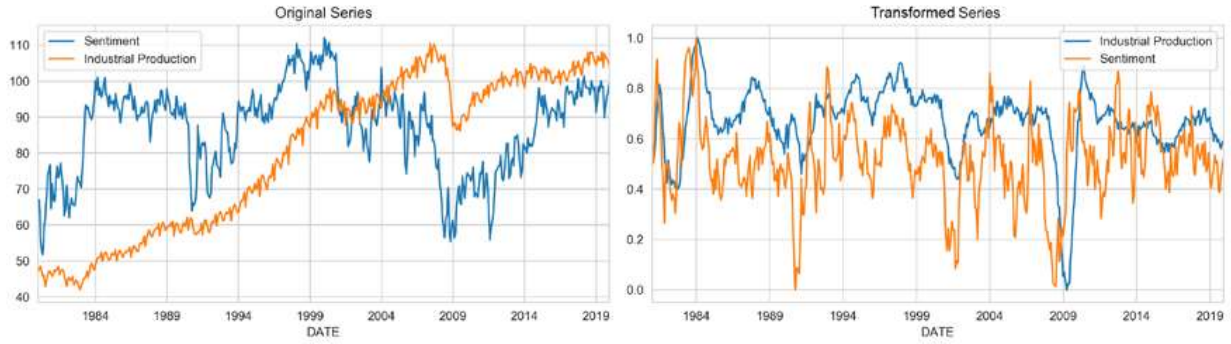


Figure 19.12: Original and transformed time series

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:

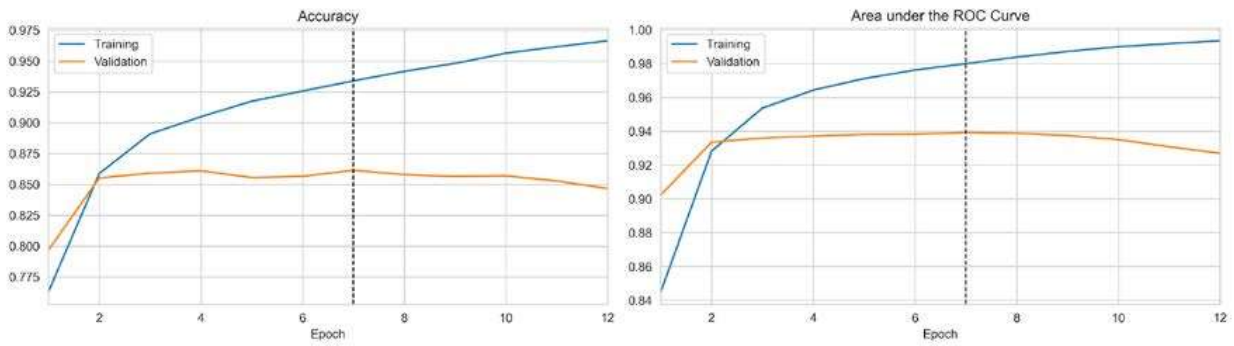


Figure 19.14: Cross-validation for RNN using IMDB data with custom embeddings

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:

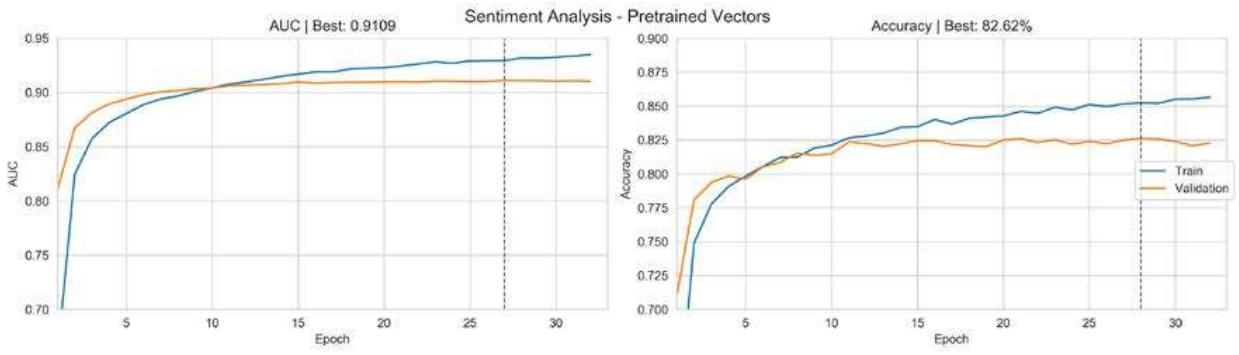


Figure 19.15: Cross-validation and test results for RNNs with multiple macro series

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 20,0 due to truncation:

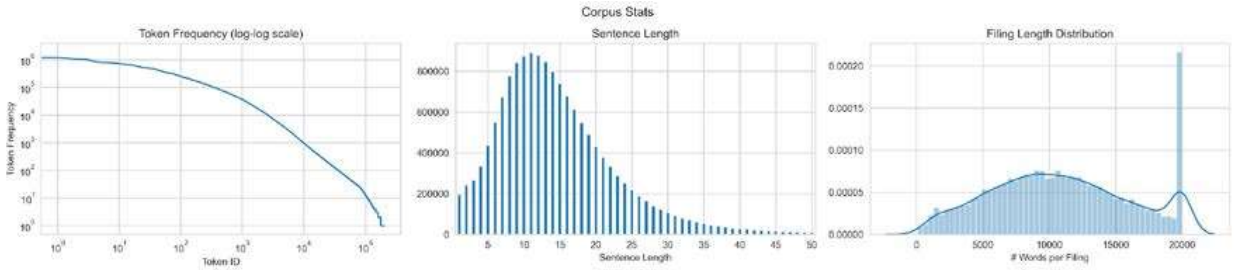


Figure 19.16: Cross-validation and test results for RNNs with multiple macro series

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:*

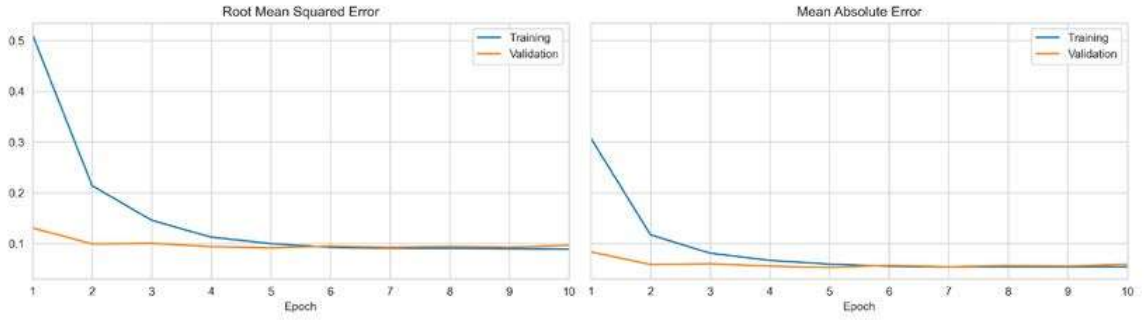


Figure 19.17: Cross-validation test results for RNNs using SEC filings to predict weekly returns

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.