17
Deep Learning for Trading

This chapter kicks off Part 4, which covers how several **deep learning (DL)** modeling techniques can be useful for investment and trading. DL has achieved numerous **breakthroughs in many domains**, ranging from image and speech recognition to robotics and intelligent agents that have drawn widespread attention and revived large-scale research into **artificial intelligence (AI)**. The expectations are high that the rapid development will continue and many more solutions to difficult practical problems will emerge.

In this chapter, we will present **feedforward neural networks** to introduce key elements of working with neural networks relevant to the various DL architectures covered in the following chapters. More specifically, we will demonstrate how to train large models efficiently using the **backpropagation algorithm** and manage the risks of overfitting. We will also show how to use the popular TensorFlow 2 and PyTorch frameworks, which we will leverage throughout Part 4.

Finally, we will develop, backtest, and evaluate a trading strategy based on signals generated by a deep feedforward neural network. We will design and tune the neural network and analyze how key hyperparameter choices affect its performance.

In summary, after reading this chapter and reviewing the accompanying notebooks, you will know about:

- How DL solves AI challenges in complex domains
- Key innovations that have propelled DL to its current popularity
- How feedforward networks learn representations from data
- Designing and training deep **neural networks (NNs)** in Python
- Implementing deep NNs using Keras, TensorFlow, and PyTorch
- Building and tuning a deep NN to predict asset returns
- Designing and backtesting a trading strategy based on deep NN signals

In the following chapters, we will build on this foundation to design various architectures suitable for different investment applications with a particular focus on alternative text and image data.

These include **recurrent neural networks (RNNs)** tailored to sequential data such as time series or natural language, and **convolutional neural networks (CNNs)**, which are particularly well suited to image data but can also be used with time-series data. We will also cover deep unsupervised learning, including autoencoders and **generative adversarial networks (GANs)** as well as reinforcement learning to train agents that interactively learn from their environment.

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.

Deep learning – what's new and why it matters

The **machine learning** (**ML**) algorithms covered in *Part 2* work well on a wide variety of important problems, including on text data, as demonstrated in *Part 3*. They have been less successful, however, in solving central AI problems such as recognizing speech or classifying objects in images. These limitations have motivated the development of DL, and the recent DL breakthroughs have greatly contributed to a resurgence of interest in AI. For a comprehensive introduction that includes and expands on many of the points in this section, see Goodfellow, Bengio, and Courville (2016), or for a much shorter version, see LeCun, Bengio, and Hinton (2015).

In this section, we outline how DL overcomes many of the limitations of other ML algorithms. These limitations particularly constrain performance on high-dimensional and unstructured data that requires sophisticated efforts to extract informative features.

The ML techniques we covered in *Parts 2* and *3* are best suited for processing structured data with well-defined features. We saw, for example, how to convert text data into tabular data using the document-text matrix in *Chapter 14*, *Text Data for Trading – Sentiment Analysis*. DL overcomes the **challenge of designing informative features**, possibly by hand, by learning a representation of the data that better captures its characteristics with respect to the outcome.

More specifically, we'll see how DL learns a **hierarchical representation of the data**, and why this approach works well for high-dimensional, unstructured data. We will describe how NNs employ a multilayered, deep architecture to compose a set of nested functions and discover a hierarchical structure. These functions compute successive and increasingly abstract representations of the data in each layer based on the learning of the previous layer. We will also look at how the backpropagation algorithm adjusts the network parameters so that these representations best meet the model's objective.

We will also briefly outline how DL fits into the evolution of AI and the diverse set of approaches that aim to achieve the current goals of AI.

Hierarchical features tame high-dimensional data

As discussed throughout *Part 2*, the key challenge of supervised learning is to generalize from training data to new samples. Generalization becomes exponentially more difficult as the dimensionality of the data increases. We encountered the root causes of these difficulties as the curse

of dimensionality in *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*.

One aspect of this curse is that volume grows exponentially with the number of dimensions: for a hypercube with edge length 10, volume increases from 10^3 to 10^4 as its dimensionality increases from three to four. Conversely, the **data density for a given sample size drops exponentially**. In other words, the number of observations required to maintain a certain density grows exponentially.

Another aspect is that functional relationships between the features and the output can become more complex when they are allowed to vary across a growing number of dimensions. As discussed in *Chapter 6*, *The Machine Learning Process*, ML algorithms struggle to learn **arbitrary functions in a high-dimensional space** because the number of candidates grows exponentially while the density of the data available to infer the relationship drops simultaneously. To mitigate this problem, algorithms hypothesize that the target function belongs to a certain class and impose constraints on the search for the optimal solution within that class for the problem at hand.

Furthermore, algorithms typically assume that the output at a new point should be similar to the output at nearby training points. This prior **assumption of smoothness** or local constancy posits that the learned function will not change much in a small region, as illustrated by the k-nearest neighbor algorithm (see *Chapter 6*, *The Machine Learning Process*). However, as data density drops exponentially with a growing number of dimensions, the distance between training samples naturally rises. The notion of nearby training examples thus becomes less meaningful as the potential complexity of the target function increases.

For traditional ML algorithms, the number of parameters and required training samples is generally proportional to the number of regions in the input space that the algorithm is able to distinguish. DL is designed to overcome the challenges of learning an exponential number of regions from a limited number of training points by assuming that a hierarchy of features generates the data.

DL as representation learning

Many AI tasks like image or speech recognition require knowledge about the world. One of the key challenges is to encode this knowledge so a computer can utilize it. For decades, the development of ML systems required considerable domain expertise to transform the raw data (such as image pixels) into an internal representation that a learning algorithm could use to detect or classify patterns.

Similarly, how much value an ML algorithm adds to a trading strategy depends greatly on our ability to engineer features that represent the predictive information in the data so that the algorithm can process it. Ideally, the features capture independent drivers of the outcome, as discussed in *Chapter 4*, *Financial Feature Engineering – How to Research*

Alpha Factors, and throughout *Parts 2* and *3* when designing and evaluating factors that capture trading signals.

Rather than relying on hand-designed features, representation learning allows an ML algorithm to automatically discover the representation of the data most useful for detecting or classifying patterns. DL combines this technique with specific assumptions about the nature of the features. See Bengio, Courville, and Vincent (2013) for additional information.

How DL extracts hierarchical features from data

The core idea behind DL is that a multi-level hierarchy of features has generated the data. Consequently, a DL model encodes the prior belief that the target function is composed of a nested set of simpler functions. This assumption permits an exponential gain in the number of regions that can be distinguished for a given number of training samples.

In other words, DL is a representation learning method that extracts a hierarchy of concepts from the data. It learns this hierarchical representation by **composing simple but non-linear functions** that successively transform the representation of one level (starting with the input data) into a new representation at a higher, slightly more abstract level. By combining enough of these transformations, DL is able to learn very complex functions.

Applied to a **classification task**, for example, higher levels of representation tend to amplify the aspects of the data most helpful for discriminating objects while suppressing irrelevant sources of variation. As we will see in more detail in *Chapter 18*, *CNNs for Financial Time Series and Satellite Images*, raw image data is just a two- or three-dimensional array of pixel values. The first layer of representation typically learns features that focus on the presence or absence of edges at particular orientations and locations. The second layer often learns motifs that depend on particular edge arrangements, regardless of small variations in their positions. The following layer may assemble motifs to represent parts of relevant objects, and subsequent layers would detect objects as combinations of these parts.

The **key breakthrough of DL** is that a general-purpose learning algorithm can extract hierarchical features suitable for modeling high-dimensional, unstructured data in a way that is infinitely more scalable than human engineering. It is thus no surprise that the rise of DL parallels the large-scale availability of unstructured image or text data. To the extent that these data sources also figure prominently among alternative data, DL has become highly relevant for algorithmic trading.

Good and bad news – the universal approximation theorem

The **universal approximation theorem** formalizes the ability of NNs to capture arbitrary relationships between input and output data. George Cybenko (1989) demonstrated that single-layer NNs using sigmoid activation functions can represent any continuous function on a closed and bounded subset of R^n. Kurt Hornik (1991) further showed that it is not

the specific shape of the activation function but rather the **multilayered architecture** that enables the hierarchical feature representation, which in turn allows NNs to approximate universal functions.

However, the theorem does not help us identify the network architecture required to represent a specific target function. We will see in the last section of this chapter that there are numerous parameters to optimize, including the network's width and depth, the number of connections between neurons, and the type of activation functions.

Furthermore, the ability to represent arbitrary functions does not imply that a network can actually learn the parameters for a given function. It took over two decades for backpropagation, the most popular learning algorithm for NNs to become effective at scale. Unfortunately, given the highly nonlinear nature of the optimization problem, there is no guarantee that it will find the absolute best rather than just a relatively good solution.

How DL relates to ML and AI

AI has a long history, going back at least to the 1950s as an academic field and much longer as a subject of human inquiry, but has experienced several waves of ebbing and flowing enthusiasm since (see Nilsson, 2009, for an in-depth survey). ML is an important subfield with a long history in related disciplines such as statistics and became prominent in the 1980s. As we have just discussed, and as depicted in *Figure 17.1*, DL is a form of representation learning and is itself a subfield of ML.

The initial goal of AI was to achieve **general AI**, conceived as the ability to solve problems considered to require human-level intelligence, and to reason and draw logical conclusions about the world and automatically improve itself. AI applications that do not involve ML include knowledge bases that encode information about the world, combined with languages for logical operations.

Historically, much AI effort went into developing **rule-based systems** that aimed to capture expert knowledge and decision-making rules, but hard-coding these rules frequently failed due to excessive complexity. In contrast, ML implies a **probabilistic approach** that learns rules from data and aims at circumventing the limitations of human-designed rule-based systems. It also involves a shift to narrower, task-specific objectives.

The following figure sketches the relationship between the various AI subfields, outlines their goals, and highlights their relevance on a timeline.

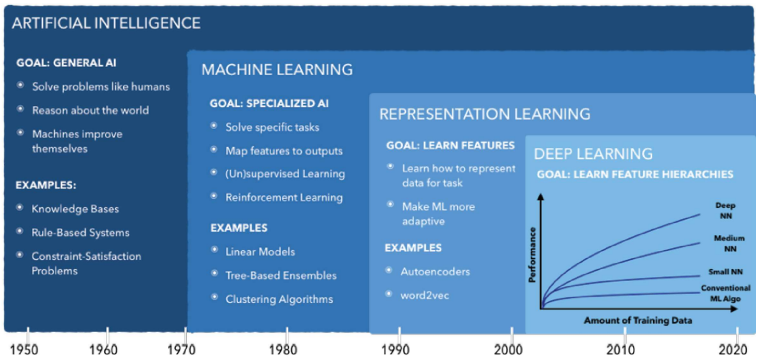

Figure 17.1: AI timeline and subfields

In the next section, we will see how to actually build a neural network.

Designing an NN

DL relies on **NNs**, which consist of a few key building blocks, which in turn can be configured in a multitude of ways. In this section, we introduce how NNs work and illustrate their most important components used to design different architectures.

(Artificial) NNs were originally inspired by biological models of learning like the human brain, either in an attempt to mimic how it works and achieve similar success, or to gain a better understanding through simulation. Current NN research draws less on neuroscience, not least since our understanding of the brain has not yet reached a sufficient level of granularity. Another constraint is overall size: even if the number of neurons used in NNs continued to double every year since their inception in the 1950s, they would only reach the scale of the human brain around 2050.

We will also explain how **backpropagation**, often simply called **backprop**, uses gradient information (the value of the partial derivative of the cost function with respect to a parameter) to adjust all neural network parameters based on training errors. The composition of various nonlinear modules implies that the optimization of the objective function can be quite challenging. We also introduce refinements of backpropagation that aim to accelerate the learning process.

A simple feedforward neural network architecture

In this section, we introduce **feedforward NNs**, which are based on the **multilayer perceptron** (**MLP**) and consist of one or more hidden layers that connect the input to the output layer. In feedforward NNs, information only flows from input to output, such that they can be represented as directed acyclic graphs, as in the following figure. In contrast, **recurrent neural networks** (**RNNs**; see *Chapter 19*, *RNNs for Multivariate Time Series and Sentiment Analysis*) include loops from the output back to the input to track or memorize past patterns and events.

We will first describe the feedforward NN architecture and how to implement it using NumPy. Then we will explain how backpropagation learns the NN weights and implement it in Python to train a binary classification network that produces perfect results even though the classes are not linearly separable. See the notebook `build_and_train_feedforward_nn` for implementation details.

A feedforward NN consists of several **layers**, each of which receives a sample of input data and produces an output. The **chain of transformations** starts with the input layer, which passes the source data to one of several internal or hidden layers, and ends with the output layer, which computes a result for comparison with the sample's output value.

The hidden and output layers consist of nodes or neurons. Nodes of a **fully connected** or dense layer connect to some or all nodes of the previous layer. The network architecture can be summarized by its depth, measured by the number of hidden layers, or the width and the number of nodes of each layer.

Each connection has a **weight** used to compute a linear combination of the input values. A layer may also have a **bias** node that always outputs a 1 and is used by the nodes in the subsequent layer, like a constant in linear regression. The goal of the training phase is to learn values for these weights that optimize the network's predictive performance.

Each node of the hidden layers computes the **dot product** of the weights and the output of the previous layer. An **activation function** transforms the result, which becomes the input to the subsequent layer. This transformation is typically nonlinear (like the sigmoid function used for logistic regression; see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts,* on linear models) so that the network can learn nonlinear relationships; we'll discuss common activation functions in the next section. The output layer computes the linear combination of the output of the last hidden layer with its weights and uses an activation function that matches the type of ML problem.

The computation of the network output from the inputs thus flows through a chain of nested functions and is called **forward propagation**. *Figure 17.2* illustrates a single-layer feedforward NN with a two-dimensional input vector, a hidden layer of width three, and two nodes in the output layer. This architecture is simple enough, so we can still easily graph it yet illustrate the key concepts.

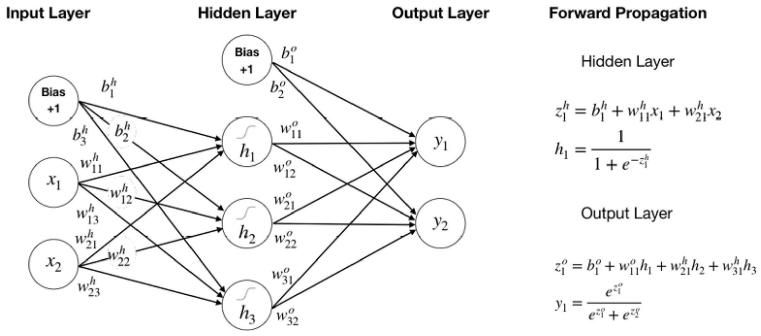

Figure 17.2: A feedforward architecture with one hidden layer

The **network graph** shows that each of the three hidden layer nodes (not counting the bias) has three weights, one for the input layer bias and two for each of the two input variables. Similarly, each output layer node has four weights to compute the product sum or dot product of the hidden layer bias and activations. In total, there are 17 parameters to be learned.

The **forward propagation** panel on the right of the figure lists the computations for an example node at the hidden and output layers, h and o, respectively. The first node in the hidden layer applies the sigmoid function to the linear combination z of its weights and inputs akin to logistic regression. The hidden layer thus runs three logistic regressions in parallel, while the backpropagation algorithm ensures that their parameters will most likely differ to best inform subsequent layers.

The output layer uses a **softmax** activation function (see *Chapter 6*, *The Machine Learning Process*) that generalizes the logistic sigmoid function to multiple classes. It adjusts the dot product of the hidden layer output with its weight to represent probabilities for the classes (only two in this case to simplify the presentation).

The forward propagation can also be expressed as nested functions, where h again represents the hidden layer and o the output layer to produce the NN estimate of the output: $\hat{y} = o\big(h(x)\big)$.

Key design choices

Some NN design choices resemble those for other supervised learning models. For example, the output is dictated by the type of the ML problem such as regression, classification, or ranking. Given the output, we need to select a cost function to measure prediction success and failure, and an algorithm that optimizes the network parameters to minimize the cost.

NN-specific choices include the numbers of layers and nodes per layer, the connections between nodes of different layers, and the type of activation functions.

A key concern is **training efficiency**: the functional form of activations can facilitate or hinder the flow of the gradient information available to the backpropagation algorithm that adjusts the weights in response to training errors. Functions with flat regions for large input value ranges have a very low gradient and can impede training progress when parameter values get stuck in such a range.

Some architectures add **skip connections** that establish direct links beyond neighboring layers to facilitate the flow of gradient information. On the other hand, the deliberate omission of connections can reduce the number of parameters to limit the network's capacity and possibly lower the generalization error, while also cutting the computational cost.

Hidden units and activation functions

Several nonlinear activation functions besides the sigmoid function have been used successfully. Their design remains an area of research because they are the key element that allows the NN to learn nonlinear relationships. They also have a critical impact on the training process because their derivatives determine how errors translate into weight adjustments.

A very popular activation function is the **rectified linear unit** (**ReLU**). The activation is computed as $g(z) = \max(0, z)$ for a given activation z, resulting in a functional form similar to the payoff for a call option. The derivative is constant whenever the unit is active. ReLUs are usually combined with an affine input transformation that requires the presence of a bias node. Their discovery has greatly improved the performance of feedforward networks compared to sigmoid units, and they are often recommended as the default. There are several ReLU extensions that aim to address the limitations of ReLU to learn via gradient descent when they are not active and their gradient is zero (Goodfellow, Bengio, and Courville, 2016).

Another alternative to the logistic function σ is the **hyperbolic tangent function tanh**, which produces output values in the ranges [-1, 1]. They are closely related because $\tanh(z) = 2\sigma(2z) - 1$. Both functions suffer from saturation because their gradient becomes very small for very low and high input values. However, tanh often performs better because it more closely resembles the identity function so that for small activation values, the network behaves more like a linear model, which in turn facilitates training.

Output units and cost functions

The choice of NN output format and cost function depends on the type of supervised learning problem:

- **Regression problems** use a linear output unit that computes the dot product of its weights with the final hidden layer activations, typically in conjunction with mean squared error cost
- **Binary classification** uses sigmoid output units to model a Bernoulli distribution just like logistic regression with hidden activations as input
- **Multiclass problems** rely on softmax units that generalize the logistic sigmoid and model a discrete distribution over more than two classes, as demonstrated earlier

Binary and multiclass problems typically use cross-entropy loss, which significantly improves training efficacy compared to mean squared error (see *Chapter 6*, *The Machine Learning Process*, for additional information on loss functions).

How to regularize deep NNs

The downside of the capacity of NNs to approximate arbitrary functions is the greatly increased risk of overfitting. The best **protection against overfitting** is to train the model on a larger dataset. Data augmentation,

such as creating slightly modified versions of images, is a powerful alternative approach. The generation of synthetic financial training data for this purpose is an active research area that we will address in *Chapter 20*, *Autoencoders for Conditional Risk Factors and Asset Pricing* (see, for example, Fu et al. 2019).

As an alternative or complement to obtaining more data, regularization can help mitigate the risk of overfitting. For all models discussed so far in this book, there is some form of regularization that modifies the learning algorithm to reduce its generalization error without negatively affecting its training error. Examples include the penalties added to the ridge and lasso regression objectives and the split or depth constraints used with decision trees and tree-based ensemble models.

Frequently, regularization takes the form of a soft constraint on the parameter values that trades off some additional bias for lower variance. A common practical finding is that the model with the lowest generalization error is not the model with the exact right size of parameters, but rather a larger model that has been well regularized. Popular NN regularization techniques that can be used in combination include parameter norm penalties, early stopping, and dropout.

Parameter norm penalties

We encountered **parameter norm penalties** for lasso and ridge regression as **L1 and L2 regularization**, respectively, in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. In the NN context, parameter norm penalties similarly modify the objective function by adding a term that represents the L1 or L2 norm of the parameters, weighted by a hyperparameter that requires tuning. For NN, the bias parameters are usually not constrained, only the weights.

L1 regularization can produce sparse parameter estimates by reducing weights all the way to zero. L2 regularization, in contrast, preserves directions along which the parameters significantly reduce the cost function. Penalties or hyperparameter values can vary across layers, but the added tuning complexity quickly becomes prohibitive.

Early stopping

We encountered **early stopping** as a regularization technique in *Chapter 12*, *Boosting Your Trading Strategy*. It is perhaps the most common NN regularization method because it is both effective and simple to use: it monitors the model's performance on a validation set and stops training when the performance ceases to improve for a certain number of observations to prevent overfitting.

Early stopping can be viewed as **efficient hyperparameter selection** that automatically determines the correct amount of regularization, whereas parameter penalties require hyperparameter tuning to identify the ideal weight decay. Just be careful to avoid **lookahead bias**: backtest results will be exceedingly positive when early stopping uses out-of-sam-

ple data that would not be available during a real-life implementation of the strategy.

Dropout

Dropout refers to the randomized omission of individual units with a given probability during forward or backward propagation. As a result, these omitted units do not contribute to the training error or receive updates.

The technique is computationally inexpensive and does not constrain the choice of model or training procedure. While more iterations are necessary to achieve the same amount of learning, each iteration is faster due to the lower computational cost. Dropout reduces the risk of overfitting by preventing units from compensating for mistakes made by other units during the training process.

Training faster – optimizations for deep learning

Backprop refers to the computation of the gradient of the cost function with respect to the internal parameter we wish to update and the use of this information to update the parameter values. The gradient is useful because it indicates the direction of parameter change that causes the maximal increase in the cost function. Hence, adjusting the parameters according to the negative gradient produces an optimal cost reduction, at least for a region very close to the observed samples. See Ruder (2017) for an excellent overview of key gradient descent optimization algorithms.

Training deep NNs can be time-consuming due to the nonconvex objective function and the potentially large number of parameters. Several challenges can significantly delay convergence, find a poor optimum, or cause oscillations or divergence from the target:

- **Local minima** can prevent convergence to a global optimum and cause poor performance
- **Flat regions with low gradients** that are not a local minimum can also prevent convergence while most likely being distant from the global optimum
- **Steep regions with high gradients** resulting from multiplying several large weights can cause excessive adjustments
- Deep architectures or long-term dependencies in an RNN require the multiplication of many weights during backpropagation, leading to **vanishing gradients** so that at least parts of the NN receive few or no updates

Several algorithms have been developed to address some of these challenges, namely variations of stochastic gradient descent and approaches that use adaptive learning rates. There is no single best algorithm, although adaptive learning rates have shown some promise.

Stochastic gradient descent

Gradient descent iteratively adjusts these parameters using the gradient information. For a given parameter θ, the basic gradient descent rule adjusts the value by the negative gradient of the loss function with respect to this parameter, multiplied by a learning rate η:

$$\theta = \theta - \underbrace{\eta}_{\text{Learning Rate}} \cdot \underbrace{\nabla_\theta J(\theta)}_{\text{Gradient}}$$

The gradient can be evaluated for all training data, a randomized batch of data, or individual observations (called online learning). Random samples give rise to **stochastic gradient descent** (**SGD**), which often leads to faster convergence if random samples are an unbiased estimate of the gradient direction throughout the training process.

However, there are numerous challenges: it can be difficult to define a learning rate or a rate schedule that facilitates efficient convergence ex ante—too low a rate prolongs the process, and too high a rate can lead to repeated overshooting and oscillation around or even divergence from a minimum. Furthermore, the same learning rate may not be adequate for all parameters, that is, in all directions of change.

Momentum

A popular refinement of basic gradient descent adds momentum to **accelerate the convergence to a local minimum**. Illustrations of momentum often use the example of a local optimum at the center of an elongated ravine (while in practice the dimensionality would be much higher than three). It implies a minimum inside a deep and narrow canyon with very steep walls that have a large gradient on one side and a much gentler slope towards a local minimum at the bottom of this region on the other side. Gradient descent naturally follows the steep gradient and will make repeated adjustments up and down the walls of the canyons with much slower movements towards the minimum.

Momentum aims to address such a situation by **tracking recent directions** and adjusting the parameters by a weighted average of the most recent gradient and the currently computed value. It uses a momentum term γ to weigh the contribution of the latest adjustment to this iteration's update v_t:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

Nesterov momentum is a simple change to normal momentum. Here, the gradient term is not computed at the current parameter space position θ_t but instead from an intermediate position. The goal is to correct for the momentum term overshooting or pointing in the wrong direction (Sutskever et al. 2013).

Adaptive learning rates

The choice of the appropriate learning rate is very challenging as highlighted in the previous subsection on stochastic gradient descent. At the same time, it is one of the most important parameters that strongly impacts training time and generalization performance.

While momentum addresses some of the issues with learning rates, it does so at the expense of introducing another hyperparameter, the **momentum rate**. Several algorithms aim to adapt the learning rate throughout the training process based on gradient information.

AdaGrad

AdaGrad accumulates all historical, parameter-specific gradient information and continues to rescale the learning rate inversely proportional to the squared cumulative gradient for a given parameter. The goal is to slow down changes for parameters that have already changed a lot and to encourage adjustments for those that haven't.

AdaGrad is designed to perform well on convex functions and has had a mixed performance in a DL context because it can reduce the learning rate too quickly based on early gradient information.

RMSProp

RMSProp modifies AdaGrad to use an exponentially weighted average of the cumulative gradient information. The goal is to put more emphasis on recent gradients. It also introduces a new hyperparameter that controls the length of the moving average.

RMSProp is a popular algorithm that often performs well, provided by the various libraries that we will introduce later and routinely used in practice.

Adam

Adam stands for **adaptive moment derivation** and combines aspects of RMSProp with Momentum. It is considered fairly robust and often used as the default optimization algorithm (Kingma and Ba, 2014).

Adam has several hyperparameters with recommended default values that may benefit from some tuning:

- **alpha**: The learning rate or step size determines how much weights are updated so that larger (smaller) values speed up (slow down) learning before the rate is updated; many libraries use the 0.001 default
- **beta**$_1$: The exponential decay rate for the first moment estimates; typically set to 0.9
- **beta**$_2$. The exponential decay rate for the second-moment estimates; usually set to 0.999
- **epsilon**: A very small number to prevent division by zero; often set to 1e-8

Summary – how to tune key hyperparameters

Hyperparameter optimization aims at **tuning the capacity of the model** so that it matches the complexity of the relationship between the input of the data. Excess capacity makes overfitting likely and requires either more data that introduces additional information into the learning process, reducing the size of the model, or more aggressive use of the various regularization tools just described.

The **principal diagnostic tool** is the behavior of training and validation error described in *Chapter 6*, *The Machine Learning Process*: if the validation error worsens while the training error continues to drop, the model is overfitting because its capacity is too high. On the other hand, if performance falls short of expectations, increasing the size of the model may be called for.

The most important aspect of parameter optimization is the architecture itself as it largely determines the number of parameters: other things being equal, more or wider hidden layers increase the capacity. As mentioned before, the best performance is often associated with models that have excess capacity but are well regularized using mechanisms like dropout or L1/L2 penalties.

In addition to **balancing model size and regularization**, it is important to tune the **learning rate** because it can undermine the optimization process and reduce the effective model capacity. The adaptive optimization algorithms offer a good starting point as described for Adam, the most popular option.

A neural network from scratch in Python

To gain a better understanding of how NNs work, we will formulate the single-layer architecture and forward propagation computations displayed in *Figure 17.2* using matrix algebra and implement it using NumPy. You can find the code samples in the notebook `build_and_train_feedforward_nn` .

The input layer

The architecture shown in *Figure 17.2* is designed for two-dimensional input data X that represents two different classes Y. In matrix form, both X and Y are of shape $N \times 2$:

$$X = \begin{bmatrix} x_{11} & x_{12} \\ \vdots & \vdots \\ x_{N1} & x_{N2} \end{bmatrix} \qquad Y = \begin{bmatrix} y_{11} & y_{12} \\ \vdots & \vdots \\ y_{N1} & y_{N2} \end{bmatrix}$$

We will generate 50,000 random binary samples in the form of two concentric circles with different radius using scikit-learn's `make_circles` function so that the classes are not linearly separable:

```
N = 50000
factor = 0.1
noise = 0.1
X, y = make_circles(n_samples=N, shuffle=True,
                    factor=factor, noise=noise)
```

We then convert the one-dimensional output into a two-dimensional array:

```
Y = np.zeros((N, 2))
for c in [0, 1]:
   Y[y == c, c] = 1
'Shape of: X: (50000, 2) | Y: (50000, 2) | y: (50000,)'
```

Figure 17.3 shows a scatterplot of the data that is clearly not linearly separable:

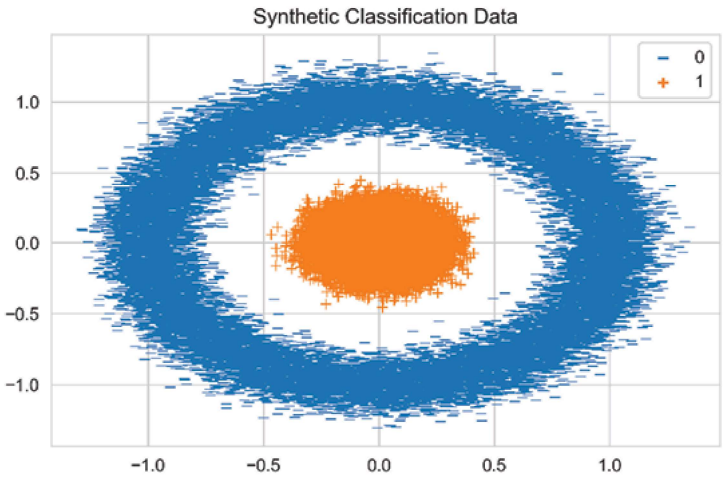

Figure 17.3: Synthetic data for binary classification

The hidden layer

The hidden layer h projects the two-dimensional input into a three-dimensional space using the weights W^h and translates the result by the bias vector b^h. To perform this affine transformation, the hidden layer weights are represented by a 2×3 matrix W^h, and the hidden layer bias vector by a three-dimensional vector:

$$\underset{2\times 3}{\mathbf{W}^h} = \begin{bmatrix} {w^h}_{11} & {w^h}_{12} & {w^h}_{13} \\ {w^h}_{21} & {w^h}_{22} & {w^h}_{23} \end{bmatrix} \quad \underset{1\times 3}{\mathbf{b}^h} \begin{bmatrix} {b^h}_1 & {b^h}_2 & {b^h}_3 \end{bmatrix}$$

The hidden layer activations H result from the application of the sigmoid function to the dot product of the input data and the weights after adding the bias vector:

$$\underset{N \times 3}{\mathbf{H}} = \sigma\left(\mathrm{X} \cdot \mathrm{W}^h + b^h\right) = \frac{1}{1 + e^{-\left(\mathrm{X} \cdot \mathrm{W}^h + b^h\right)}} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ \vdots & \vdots & \vdots \\ h_{N1} & h_{N2} & h_{N3} \end{bmatrix}$$

To implement the hidden layer using NumPy, we first define the `logistic` sigmoid function:

```
def logistic(z):
    """Logistic function."""
    return 1 / (1 + np.exp(-z))
```

We then define a function that computes the hidden layer activations as a function of the relevant inputs, weights, and bias values:

```
def hidden_layer(input_data, weights, bias):
    """Compute hidden activations"""
    return logistic(input_data @ weights + bias)
```

The output layer

The output layer compresses the three-dimensional hidden layer activations H back to two dimensions using a 3×2 weight matrix W^o and a two-dimensional bias vector $\mathbf{b}^o$:

$$\underset{3 \times 2}{\mathbf{W}^o} = \begin{bmatrix} \mathrm{w}^o{}_{11} & \mathrm{w}^o{}_{12} \\ \mathrm{w}^o{}_{21} & \mathrm{w}^o{}_{22} \\ \mathrm{w}^o{}_{31} & \mathrm{w}^o{}_{32} \end{bmatrix} \qquad \underset{1 \times 2}{\mathbf{b}^o} = \begin{bmatrix} \mathrm{b}^o{}_1 & \mathrm{b}^o{}_2 \end{bmatrix}$$

The linear combination of the hidden layer outputs results in an $N \times 2$ matrix $\mathbf{Z}^o$:

$$\underset{N \times 2}{\mathbf{Z}^o} = \underset{N \times 3}{\mathbf{H}} \cdot \underset{3 \times 2}{\mathbf{W}^o} + \underset{1 \times 2}{\mathbf{b}^o}$$

The output layer activations are computed by the softmax function ς that normalizes the $\mathbf{Z}^o$ to conform to the conventions used for discrete probability distributions:

$$\underset{N \times 2}{\mathbf{Y}} = \varsigma\left(\mathbf{H} \cdot \mathbf{W}^o + \mathbf{b}^o\right) = \begin{bmatrix} y_{11} & y_{12} \\ \vdots & \vdots \\ y_{n1} & y_{n2} \end{bmatrix}$$

We create a softmax function in Python as follows:

```
def softmax(z):
    """Softmax function"""
```

```
    return np.exp(z) / np.sum(np.exp(z), axis=1, keepdims=True)
```

As defined here, the output layer activations depend on the hidden layer activations and the output layer weights and biases:

```
def output_layer(hidden_activations, weights, bias):
    """Compute the output y_hat"""
    return softmax(hidden_activations @ weights + bias)
```

Now we have all the components we need to integrate the layers and compute the NN output directly from the input.

Forward propagation

The `forward_prop` function combines the previous operations to yield the output activations from the input data as a function of weights and biases:

```
def forward_prop(data, hidden_weights, hidden_bias, output_weights, output_bias):
    """Neural network as function."""
    hidden_activations = hidden_layer(data, hidden_weights, hidden_bias)
    return output_layer(hidden_activations, output_weights, output_bias)
```

The `predict` function produces the binary class predictions given weights, biases, and input data:

```
def predict(data, hidden_weights, hidden_bias, output_weights, output_bias):
    """Predicts class 0 or 1"""
    y_pred_proba = forward_prop(data,
                                hidden_weights,
                                hidden_bias,
                                output_weights,
                                output_bias)
    return np.around(y_pred_proba)
```

The cross-entropy cost function

The final piece is the cost function to evaluate the NN output based on the given label. The cost function J uses the cross-entropy loss ξ, which sums the deviations of the predictions for each class c from the actual outcome:

$$J\left(\mathbf{Y}, \hat{\mathbf{Y}}\right) = \sum_{i=1}^{n} \xi(y_i, \hat{y}_i) = -\sum_{i=1}^{N} \sum_{i=c}^{C} y_{ic} \cdot log(\hat{y}_{ic})$$

It takes the following form in Python:

```
def loss(y_hat, y_true):
    """Cross-entropy"""
    return - (y_true * np.log(y_hat)).sum()
```

How to implement backprop using Python

To update the NN weights and bias values using backprop, we need to compute the gradient of the cost function. The gradient represents the partial derivative of the cost function with respect to the target parameter.

How to compute the gradient

The NN composes a set of nested functions as highlighted earlier. Hence, the gradient of the loss function with respect to internal, hidden parameters is computed using the chain rule of calculus.

For scalar values, given the functions $z = h(x)$ and $y = o(h(x)) = o(z)$, we compute the derivative of y with respect to x using the chain rule as follows:

$$\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx}$$

For vectors, with $z \in \mathrm{R}^m$ and $x \in \mathrm{R}^n$ so that the hidden layer h maps from R^n to R^m and $z = h(x)$ and $y = o(z)$, we get:

$$\frac{\partial y}{\partial x_i} = \sum_j \frac{\partial y}{\partial z_j}\frac{\partial z_j}{\partial x_i}$$

We can express this more concisely using matrix notation using the $m \times n$ Jacobian matrix of h:

$$m \times n \frac{dz}{dx}$$

which contains the partial derivatives for each of the m components of z with respect to each of the n inputs x. The gradient ∇ of y with respect to x contains all partial derivatives and can thus be written as:

$$\nabla_{\mathbf{x}} y = \left(\frac{dz}{dx}\right)^T \nabla_{\mathbf{z}} y$$

The loss function gradient

The derivative of the cross-entropy loss function J with respect to each output layer activation $i = 1, ..., N$ is a very simple expression (see the notebook for details), shown below on the left for scalar values and on the right in matrix notation:

$$\frac{\partial J}{\partial z_i^0} = \hat{y}_i - y_i \qquad \nabla_{\mathbf{z}^0} J = \hat{\mathbf{Y}} - \mathbf{Y} = \delta^0$$

We define `loss_gradient` function accordingly:

```
def loss_gradient(y_hat, y_true):
    """output layer gradient"""
    return y_hat - y_true
```

The output layer gradients

To propagate the update back to the output layer weights, we use the gradient of the loss function J with respect to the weight matrix:

$$\frac{\partial J}{\partial \mathbf{W}^0} = H^T \cdot \left(\hat{\mathbf{Y}} - \mathbf{Y}\right) = H^T \cdot \delta^0$$

and for the bias:

$$\frac{\partial J}{\partial \mathbf{b}^0} = \frac{\partial J}{\partial Y} \frac{\partial Y}{\partial Z^0} \frac{\partial Z^0}{\partial \mathbf{b}^0} = \sum_{i=1}^{N} 1 \cdot (\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i) = \sum_{i=1}^{N} \delta_i{}^0$$

We can now define `output_weight_gradient` and `output_bias_gradient` accordingly, both taking the loss gradient δ^0 as input:

```
def output_weight_gradient(H, loss_grad):
    """Gradients for the output layer weights"""
    return  H.T @ loss_grad
def output_bias_gradient(loss_grad):
    """Gradients for the output layer bias"""
    return np.sum(loss_grad, axis=0, keepdims=True)
```

The hidden layer gradients

The gradient of the loss function with respect to the hidden layer values computes as follows, where $\circ$ refers to the element-wise matrix product:

$$\nabla_{\mathbf{z}^h} J = \mathbf{H} \circ (1 - \mathbf{H}) \circ [\delta_0 \cdot (\mathbf{W}^0)^T] = \delta_h$$

We define a `hidden_layer_gradient` function to encode this result:

```
def hidden_layer_gradient(H, out_weights, loss_grad):
    """Error at the hidden layer.
    H * (1-H) * (E . Wo^T)"""
    return H * (1 - H) * (loss_grad @ out_weights.T)
```

The gradients for hidden layer weights and biases are:

$$\nabla_{\mathbf{W}^h} J = \mathbf{X}^T \cdot \delta^h \qquad \nabla_{\mathbf{b}^h} J = \sum_{j=1}^{N} \delta_{hj}$$

The corresponding functions are:

```
def hidden_weight_gradient(X, hidden_layer_grad):
    """Gradient for the weight parameters at the hidden layer"""
    return X.T @ hidden_layer_grad

def hidden_bias_gradient(hidden_layer_grad):
    """Gradient for the bias parameters at the output layer"""
    return np.sum(hidden_layer_grad, axis=0, keepdims=True)
```

Putting it all together

To prepare for the training of our network, we create a function that combines the previous gradient definition and computes the relevant weight and bias updates from the training data and labels, and the current weight and bias values:

```
def compute_gradients(X, y_true, w_h, b_h, w_o, b_o):
    """Evaluate gradients for parameter updates"""
    # Compute hidden and output layer activations
    hidden_activations = hidden_layer(X, w_h, b_h)
    y_hat = output_layer(hidden_activations, w_o, b_o)
    # Compute the output layer gradients
    loss_grad = loss_gradient(y_hat, y_true)
    out_weight_grad = output_weight_gradient(hidden_activations, loss_grad)
    out_bias_grad = output_bias_gradient(loss_grad)
    # Compute the hidden layer gradients
    hidden_layer_grad = hidden_layer_gradient(hidden_activations,
                                              w_o, loss_grad)
    hidden_weight_grad = hidden_weight_gradient(X, hidden_layer_grad)
    hidden_bias_grad = hidden_bias_gradient(hidden_layer_grad)
    return [hidden_weight_grad, hidden_bias_grad, out_weight_grad, out_bias_grad]
```

Testing the gradients

The notebook contains a test function that compares the gradient derived previously analytically using multivariate calculus to a numerical estimate that we obtain by slightly perturbing individual parameters. The

test function validates that the resulting change in output value is similar to the change estimated by the analytical gradient.

Implementing momentum updates using Python

To incorporate momentum into the parameter updates, define an `update_momentum` function that combines the results of the `compute_gradients` function we just used with the most recent momentum updates for each parameter matrix:

```
def update_momentum(X, y_true, param_list, Ms, momentum_term, learning_rate):
    """Compute updates with momentum."""
    gradients = compute_gradients(X, y_true, *param_list)
    return [momentum_term * momentum - learning_rate * grads
            for momentum, grads in zip(Ms, gradients)]
```

The `update_params` function performs the actual updates:

```
def update_params(param_list, Ms):
    """Update the parameters."""
    return [P + M for P, M in zip(param_list, Ms)]
```

Training the network

To train the network, we first randomly initialize all network parameters using a standard normal distribution (see the notebook). For a given number of iterations or epochs, we run momentum updates and compute the training loss as follows:

```
def train_network(iterations=1000, lr=.01, mf=.1):
    # Initialize weights and biases
    param_list = list(initialize_weights())
    # Momentum Matrices = [MWh, Mbh, MWo, Mbo]
    Ms = [np.zeros_like(M) for M in param_list]
    train_loss = [loss(forward_prop(X, *param_list), Y)]
    for i in range(iterations):
        # Update the moments and the parameters
        Ms = update_momentum(X, Y, param_list, Ms, mf, lr)
        param_list = update_params(param_list, Ms)
        train_loss.append(loss(forward_prop(X, *param_list), Y))
    return param_list, train_loss
```

Figure 17.4 plots the training loss over 50,000 iterations for 50,000 training samples with a momentum term of 0.5 and a learning rate of 1e-4. It shows that it takes over 5,000 iterations for the loss to start to decline but then does so very fast. We have not used SGD, which would have likely accelerated convergence significantly.

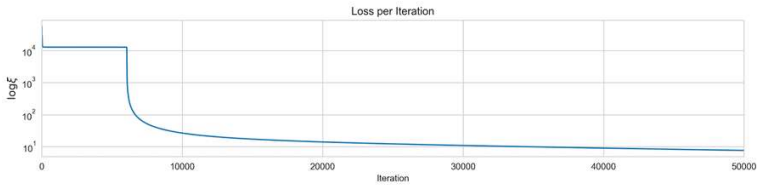

Figure 17.4: Training loss per iteration

The plots in *Figure 17.5* show the function learned by the neural network with a three-dimensional hidden layer from two-dimensional data with two classes that are not linearly separable. The left panel displays the source data and the decision boundary that misclassifies very few data points and would further improve with continued training.

The center panel shows the representation of the input data learned by the hidden layer. The network learns weights so that the projection of the input from two to three dimensions enables the linear separation of the two classes. The right plot shows how the output layer implements the linear separation in the form of a cutoff value of 0.5 in the output dimension:

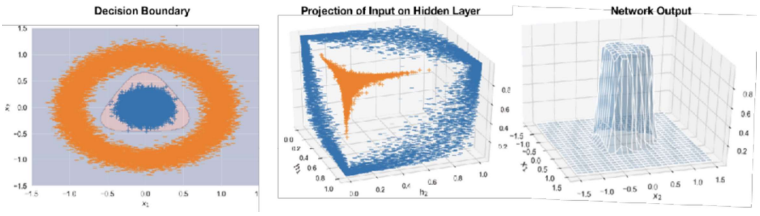

Figure 17.5: Visualizing the function learned by the neural network

To **sum up**: we have seen how a very simple network with a single hidden layer with three nodes and a total of 17 parameters is able to learn how to solve a nonlinear classification problem using backprop and gradient descent with momentum.

We will next review how to use popular DL libraries that facilitate the design and fast training of complex architectures while using sophisticated techniques to prevent overfitting and evaluate the results.

Popular deep learning libraries

Currently, the most popular DL libraries are TensorFlow (supported by Google), Keras (led by Francois Chollet, now at Google), and PyTorch (supported by Facebook). Development is very active with PyTorch at version 1.4 and TensorFlow at 2.2 as of March 2020. TensorFlow 2.0 adopted Keras as its main interface, effectively combining both libraries into one.

All libraries provide the design choices, regularization methods, and backprop optimizations we discussed previously in this chapter. They also facilitate fast training on one or several **graphics processing units (GPUs)**. The libraries differ slightly in their focus with TensorFlow origi-

nally designed for deployment in production and prevalent in the industry, while PyTorch has been popular among academic researchers; however, the interfaces are gradually converging.

We will illustrate the use of TensorFlow and PyTorch using the same network architecture and dataset as in the previous section.

Leveraging GPU acceleration

DL is very computationally intensive, and good results often require large datasets. As a result, model training and evaluation can become rather time-consuming. GPUs are highly optimized for the matrix operations required by deep learning models and tend to have more processing power, rendering speedups of 10x or more not uncommon.

All popular deep learning libraries support the use of a GPU, and some also allow for parallel training on multiple GPUs. The most common types of GPU are produced by NVIDIA, and configuration requires installation and setup of the CUDA environment. The process continues to evolve and can be somewhat challenging depending on your computational environment.

A more straightforward way to leverage GPU is via the Docker virtualization platform. There are numerous images available that you can run in a local container managed by Docker that circumvents many of the driver and version conflicts that you may otherwise encounter. TensorFlow provides Docker images on its website that can also be used with Keras.

See GitHub for references and related instructions in the DL notebooks and the installation directory.

How to use TensorFlow 2

TensorFlow became the leading deep learning library shortly after its release in September 2015, one year before PyTorch. TensorFlow 2 simplified the API that had grown increasingly complex over time by making the Keras API its principal interface.

Keras was designed as a high-level API to accelerate the iterative workflow of designing and training deep neural networks with computational backends like TensorFlow, Theano, or CNTK. It has been integrated into TensorFlow in 2017. You can also combine code from both libraries to leverage Keras' high-level abstractions as well as customized TensorFlow graph operations.

In addition, TensorFlow adopts **eager execution**. Previously, you needed to define a complete computational graph for compilation into optimized operations. Running the compiled graph required the configuration of a session and the provision of the requisite data. Under eager execution, you can run TensorFlow operations on a line-by-line basis just like common Python code.

Keras supports both a slightly simpler Sequential API and a more flexible Functional API. We will introduce the former at this point and use the Functional API in more complex examples in the following chapters.

To create a model, we just need to instantiate a `Sequential` object and provide a list with the sequence of standard layers and their configurations, including the number of units, type of activation function, or name.

The first hidden layer needs information about the number of features in the matrix it receives from the input layer via the `input_shape` argument. In our simple case, there are just two. Keras infers the number of rows it needs to process during training, through the `batch_size` argument that we will pass to the `fit` method later in this section. TensorFlow infers the sizes of the inputs received by other layers from the previous layer's `units` argument:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
model = Sequential([
    Dense(units=3, input_shape=(2,), name='hidden'),
    Activation('sigmoid', name='logistic'),
    Dense(2, name='output'),
    Activation('softmax', name='softmax'),
])
```

The Keras API provides numerous standard building blocks, including recurrent and convolutional layers, various options for regularization, a range of loss functions and optimizers, and also preprocessing, visualization, and logging (see the link to the TensorFlow documentation on GitHub for reference). It is also extensible.

The model's `summary` method produces a concise description of the network architecture, including a list of the layer types and shapes and the number of parameters:

```
model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 3)                 9
_________________________________________________________________
logistic (Activation)        (None, 3)                 0
_________________________________________________________________
output (Dense)               (None, 2)                 8
_________________________________________________________________
softmax (Activation)         (None, 2)                 0
=================================================================
Total params: 17
Trainable params: 17
Non-trainable params: 0
```

Next, we compile the Sequential model to configure the learning process. To this end, we define the optimizer, the loss function, and one or several performance metrics to monitor during training:

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Keras uses callbacks to enable certain functionality during training, such as logging information for interactive display in TensorBoard (see the next section):

```
tb_callback = TensorBoard(log_dir='./tensorboard',
                          histogram_freq=1,
                          write_graph=True,
                          write_images=True)
```

To train the model, we call its `fit` method and pass several parameters in addition to the training data:

```
model.fit(X, Y,
          epochs=25,
          validation_split=.2,
          batch_size=128,
          verbose=1,
          callbacks=[tb_callback])
```

See the notebook for a visualization of the decision boundary that resembles the result from our earlier manual network implementation. The training with TensorFlow runs orders of magnitude faster, though.

How to use TensorBoard

TensorBoard is a great suite of visualization tools that comes with TensorFlow. It includes visualization tools to simplify the understanding, debugging, and optimization of NNs.

You can use it to visualize the computational graph, plot various execution and performance metrics, and even visualize image data processed by the network. It also permits comparisons of different training runs.

When you run the `how_to_use_tensorflow` notebook, with TensorFlow installed, you can launch TensorBoard from the command line:

```
tensorboard --logdir=/full_path_to_your_logs ## e.g. ./tensorboard
```

Alternatively, you can use it within your notebook by first loading the extension and then starting TensorBoard similarly by referencing the `log` directory:

```
%load_ext tensorboard
%tensorboard --logdir tensorboard/
```

For starters, the visualizations include train and validation metrics (see the left panel of *Figure 17.6*).

In addition, you can view histograms of the weights and biases over various epochs (right panel of Figure 17.6; epochs evolve from back to front). This is useful because it allows you to monitor whether backpropagation succeeds in adjusting the weights as learning progresses and whether they are converging.

The values of weights should change from their initialization values over the course of several epochs and eventually stabilize:

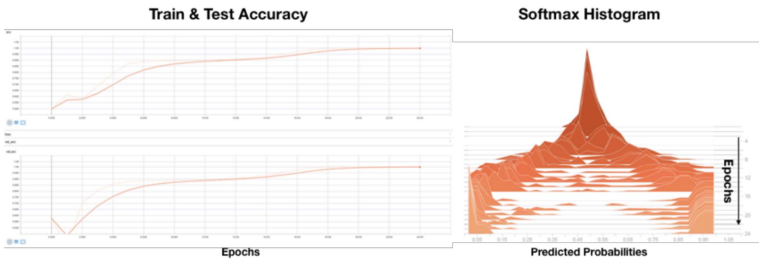

Figure 17.6: TensorBoard learning process visualization

TensorBoard also lets you display and interactively explore the computational graph of your network, drilling down from the high-level structure to the underlying operations by clicking on the various nodes. The visualization for our simple example architecture (see the notebook) already includes numerous components but is very useful when debugging. For further reference, see the links on GitHub to more detailed tutorials.

How to use PyTorch 1.4

PyTorch was developed at the **Facebook AI Research (FAIR)** group led by Yann LeCunn, and the first alpha version released in September 2016. It provides deep integration with Python libraries like NumPy that can be used to extend its functionality, strong GPU acceleration, and automatic differentiation using its autograd system. It provides more granular control than Keras through a lower-level API and is mainly used as a deep learning research platform but can also replace NumPy while enabling GPU computation.

It employs eager execution, in contrast to the static computation graphs used by, for example, Theano or TensorFlow. Rather than initially defining and compiling a network for fast but static execution, it relies on its autograd package for automatic differentiation of tensor operations; that is, it computes gradients "on the fly" so that network structures can be partially modified more easily. This is called **define-by-run**, meaning that backpropagation is defined by how your code runs, which in turn implies that every single iteration can be different. The PyTorch documentation provides a detailed tutorial on this.

The resulting flexibility combined with an intuitive Python-first interface and speed of execution has contributed to its rapid rise in popularity and

led to the development of numerous supporting libraries that extend its functionality.

Let's see how PyTorch and autograd work by implementing our simple network architecture (see the `how_to_use_pytorch` notebook for details).

How to create a PyTorch DataLoader

We begin by converting the NumPy or pandas input data to `torch` tensors. Conversion from and to NumPy is very straightforward:

```
import torch
X_tensor = torch.from_numpy(X)
y_tensor = torch.from_numpy(y)
X_tensor.shape, y_tensor.shape
(torch.Size([50000, 2]), torch.Size([50000]))
```

We can use these PyTorch tensors to instantiate first a `TensorDataset` and, in a second step, a `DataLoader` that includes information about `batch_size`:

```
import torch.utils.data as utils
dataset = utils.TensorDataset(X_tensor,y_tensor)
dataloader = utils.DataLoader(dataset,
                              batch_size=batch_size,
                              shuffle=True)
```

How to define the neural network architecture

PyTorch defines an NN architecture using the `Net()` class. The central element is the `forward` function. autograd automatically defines the corresponding `backward` function that computes the gradients.

Any legal tensor operation is fair game for the `forward` function, providing a log of design flexibility. In our simple case, we just link the tensor through functional input-output relations after initializing their attributes:

```
import torch.nn as nn
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()  # Inherited from nn.Module
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.logistic = nn.LogSigmoid()
        self.fc2 = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        """Forward pass: stacking each layer together"""
        out = self.fc1(x)
        out = self.logistic(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out
```

We then instantiate a `Net()` object and can inspect the architecture as follows:

```
net = Net(input_size, hidden_size, num_classes)
net
Net(
  (fc1): Linear(in_features=2, out_features=3, bias=True)
  (logistic): LogSigmoid()
  (fc2): Linear(in_features=3, out_features=2, bias=True)
  (softmax): Softmax()
)
```

To illustrate eager execution, we can also inspect the initialized parameters in the first tensor:

```
list(net.parameters())[0]
Parameter containing:
tensor([[ 0.3008, -0.2117],
        [-0.5846, -0.1690],
        [-0.6639,  0.1887]], requires_grad=True)
```

To enable GPU processing, you can use `net.cuda()`. See the PyTorch documentation for placing tensors on CPU and/or one or more GPU units.

We also need to define a loss function and the optimizer, using some of the built-in options:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
```

How to train the model

Model training consists of an outer loop for each epoch, that is, each pass over the training data, and an inner loop over the batches produced by the `DataLoader`. That executes the forward and backward passes of the learning algorithm. Some care needs to be taken to adjust data types to the requirements of the various objects and functions; for example, labels need to be integers and the features should be of type `float`:

```
for epoch in range(num_epochs):
    print(epoch)
    for i, (features, label) in enumerate(dataloader):

        features = Variable(features.float())
        label = Variable(label.long())
        # Initialize the hidden weights
        optimizer.zero_grad()

        # Forward pass: compute output given features
        outputs = net(features)

        # Compute the loss
        loss = criterion(outputs, label)
        # Backward pass: compute the gradients
```

```
            loss.backward()
            # Update the weights
            optimizer.step()
```

The notebook also contains an example that uses the `livelossplot` package to plot losses throughout the training process as provided by Keras out of the box.

How to evaluate the model predictions

To obtain predictions from our trained model, we pass it feature data and convert the prediction to a NumPy array. We get softmax probabilities for each of the two classes:

```
test_value = Variable(torch.from_numpy(X)).float()
prediction = net(test_value).data.numpy()
Prediction.shape
(50000, 2)
```

From here on, we can proceed as before to compute loss metrics or visualize the result that again reproduces a version of the decision boundary we found earlier.

Alternative options

The huge interest in DL has led to the development of several competing libraries that facilitate the design and training of NNs. The most prominent include the following examples (also see references on GitHub).

Apache MXNet

MXNet, incubated at the Apache Foundation, is an open source DL software framework used to train and deploy deep NNs. It focuses on scalability and fast model training. They included the Gluon high-level interface to make it easy to prototype, train, and deploy DL models. MXNet has been picked by Amazon for deep learning on AWS.

Microsoft Cognitive Toolkit (CNTK)

The Cognitive Toolkit, previously known as CNTK, is Microsoft's contribution to the deep learning library collection. It describes an NN as a series of computational steps via a directed graph, similar to TensorFlow. In this directed graph, leaf nodes represent input values or network parameters, while other nodes represent matrix operations upon their inputs. CNTK allows users to build and combine popular model architectures ranging from deep feedforward NNs, convolutional networks, and recurrent networks (RNNs/LSTMs).

Fastai

The fastai library aims to simplify training NNs that are fast and accurate using modern best practices. These practices have emerged from research into DL at the company that makes both the software and accom-

panying courses available for free. Fastai includes support for models that process image, text, tabular, and collaborative filtering data.

Optimizing an NN for a long-short strategy

In practice, we need to explore variations for the design options for the NN architecture and how we train it from those we outlined previously because we can never be sure from the outset which configuration best suits the data. In this section, we will explore various architectures for a simple feedforward NN to predict daily stock returns using the dataset developed in *Chapter 12* (see the notebook `preparing_the_model_data` in the GitHub directory for that chapter).

To this end, we will define a function that returns a TensorFlow model based on several architectural input parameters and cross-validate alternative designs using the `MultipleTimeSeriesCV` we introduced in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. To assess the signal quality of the model predictions, we build a simple ranking-based long-short strategy based on an ensemble of the models that perform best during the in-sample cross-validation period. To limit the risk of false discoveries, we then evaluate the performance of this strategy for an out-of-sample test period.

See the `optimizing_a_NN_architecture_for_trading` notebook for details.

Engineering features to predict daily stock returns

To develop our trading strategy, we use the daily stock returns for 995 US stocks for the eight-year period from 2010 to 2017. We will use the features developed in *Chapter 12, Boosting Your Trading Strategy* that include volatility and momentum factors, as well as lagged returns with cross-sectional and sectoral rankings. We load the data as follows:

```
data = pd.read_hdf('../12_gradient_boosting_machines/data/data.h5',
                   'model_data').dropna()
outcomes = data.filter(like='fwd').columns.tolist()
lookahead = 1
outcome= f'r{lookahead:02}_fwd'
X = data.loc[idx[:, :'2017'], :].drop(outcomes, axis=1)
y = data.loc[idx[:, :'2017'], outcome]
```

Defining an NN architecture framework

To automate the generation of our TensorFlow model, we create a function that constructs and compiles the model based on arguments that can later be passed during cross-validation iterations.

The following `make_model` function illustrates how to flexibly define various architectural elements for the search process. The `dense_layers` argument defines both the depth and width of the network as a list of integers. We also use `dropout` for regularization, expressed as a float in the range [0, 1] to define the probability that a given unit will be excluded from a training iteration:

```
def make_model(dense_layers, activation, dropout):
    '''Creates a multi-layer perceptron model

    dense_layers: List of layer sizes; one number per layer
    '''
    model = Sequential()
    for i, layer_size in enumerate(dense_layers, 1):
        if i == 1:
            model.add(Dense(layer_size, input_dim=X_cv.shape[1]))
            model.add(Activation(activation))
        else:
            model.add(Dense(layer_size))
            model.add(Activation(activation))
    model.add(Dropout(dropout))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error',
                  optimizer='Adam')
    return model
```

Now we can turn to the cross-validation process to evaluate various NN architectures.

Cross-validating design options to tune the NN

We use the `MultipleTimeSeriesCV` to split the data into rolling training and validation sets comprising of 24 * 12 months of data, while keeping the final 12 * 21 days of data (starting November 30, 2016) as a holdout test. We train each model for 48 21-day periods and evaluate its results over 3 21-day periods, implying 12 splits for cross-validation and test periods combined:

```
n_splits = 12
train_period_length=21 * 12 * 4
test_period_length=21 * 3
cv = MultipleTimeSeriesCV(n_splits=n_splits,
                          train_period_length=train_period_length,
                          test_period_length=test_period_length,
                          lookahead=lookahead)
```

Next, we define a set of configurations for cross-validation. These include several options for two hidden layers and dropout probabilities; we'll only use tanh activations because a trial run did not suggest significant differences compared to ReLU. (We could also try out different optimizers. but I recommend you do not run this experiment, to limit what is already a computationally intensive effort):

```
dense_layer_opts = [(16, 8), (32, 16), (32, 32), (64, 32)]
dropout_opts = [0, .1, .2]
param_grid = list(product(dense_layer_opts, activation_opts, dropout_opts))
np.random.shuffle(param_grid)
len(param_grid)
12
```

To run the cross-validation, we define a function that produces the train and validation data based on the integer indices produced by the `MultipleTimeSeriesCV` as follows:

```
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]
    return x_train, y_train, x_val, y_val
```

During cross-validation, we train a model using one set of parameters from the previously defined grid for 20 epochs. After each epoch, we store a `checkpoint` that contains the learned weights that we can reload to quickly generate predictions for the best configuration without retraining.

After each epoch, we compute and store the **information coefficient** (**IC**) for the validation set by day:

```
ic = []
scaler = StandardScaler()
for params in param_grid:
    dense_layers, activation, dropout = params
    for batch_size in [64, 256]:
        checkpoint_path = checkpoint_dir / str(dense_layers) / activation /
                          str(dropout) / str(batch_size)
        for fold, (train_idx, test_idx) in enumerate(cv.split(X_cv)):
            x_train, y_train, x_val, y_val = get_train_valid_data(X_cv, y_cv,
                                             train_idx, test_idx)
            x_train = scaler.fit_transform(x_train)
            x_val = scaler.transform(x_val)
            preds = y_val.to_frame('actual')
            r = pd.DataFrame(index=y_val.groupby(level='date').size().index)
            model = make_model(dense_layers, activation, dropout)
            for epoch in range(20):
                model.fit(x_train, y_train,
                          batch_size=batch_size,
                          epochs=1, validation_data=(x_val, y_val))
                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
            ic.append(r.assign(dense_layers=str(dense_layers),
                               activation=activation,
                               dropout=dropout,
                               batch_size=batch_size,
                               fold=fold))
```

With an NVIDIA GTX 1080 GPU, 20 epochs takes a bit over one hour with batches of 64 samples, and around 20 minutes with 256 samples.

Evaluating the predictive performance

Let's first take a look at the five models that achieved the highest median daily IC during the cross-validation period. The following code computes these values:

```
dates = sorted(ic.index.unique())
cv_period = 24 * 21
cv_dates = dates[:cv_period]
ic_cv = ic.loc[cv_dates]
(ic_cv.drop('fold', axis=1).groupby(params).median().stack()
 .to_frame('ic').reset_index().rename(columns={'level_3': 'epoch'})
 .nlargest(n=5, columns='ic'))
```

The resulting table shows that the architectures using 32 units in both layers and 16/8 in the first/second layer, respectively, performed best. These models also use `dropout` and were trained with batch sizes of 64 samples with the given number of epochs for all folds. The median IC values vary between 0.0236 and 0.0246:

Dense Layers	Dropout	Batch Size	Epoch	IC
(32, 32)	0.1	64	7	0.0246
(16, 8)	0.2	64	14	0.0241
(16, 8)	0.1	64	3	0.0238
(32, 32)	0.1	64	10	0.0237
(16, 8)	0.2	256	3	0.0236

Next, we'll take a look at how the parameter choices impact the predictive performance.

First, we visualize the daily information coefficient (averaged per fold) for different configurations by epoch to understand how the duration of training affects the predictive accuracy. The plots in *Figure 17.7*, however, highlight few conclusive patterns; the IC varies little across models and not particularly systematically across epochs:

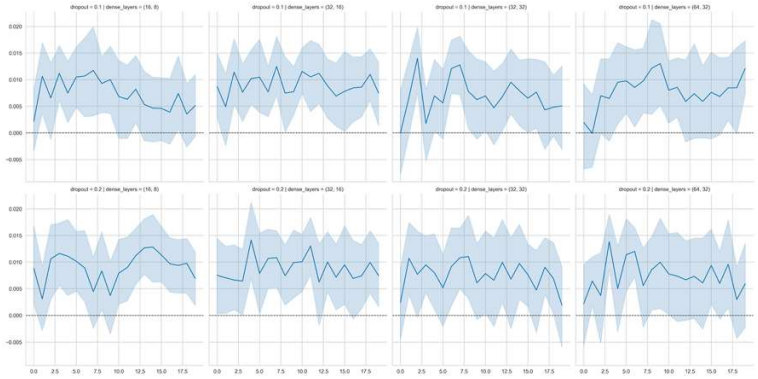

Figure 17.7: Information coefficients for various model configurations

For more statistically robust insights, we run a linear regression using **ordinary least squares (OLS)** (see *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*) using dummy variables for the layer, dropout, and batch size choices as well as for each epoch:

```
data = pd.melt(ic, id_vars=params, var_name='epoch', value_name='ic')
data = pd.get_dummies(data, columns=['epoch'] + params, drop_first=True)
model = sm.OLS(endog=data.ic, exog=sm.add_constant(data.drop('ic', axis=1)))
```

The chart in *Figure 17.8* plots the confidence interval for each regression coefficient; if it does not include zero, then the coefficient is significant at the five percent level. The IC values on the y-axis reflect the differential from the constant (0.0027, p-value: 0.017) that represents the sample average over the configuration excluded while dropping one category of each dummy variable.

Across all configurations, batch size 256 and a dropout of 0.2 made significant (but small) positive contributions to performance. Similarly, training for seven epochs yielded slightly superior results. The regression is overall significant according to the F statistic but has a very low R2 value close to zero, underlining the high degree of noise in the data relative to the signal conveyed by the parameter choices.

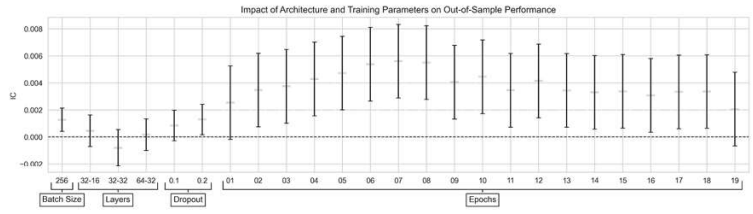

Figure 17.8: OLS coefficients and confidence intervals

Backtesting a strategy based on ensembled signals

To translate our NN model into a trading strategy, we generate predictions, evaluate their signal quality, create rules that define how to trade on these predictions, and backtest the performance of a strategy that im-

plements these rules. See the notebook `backtesting_with_zipline` for the code examples in this section.

Ensembling predictions to produce tradeable signals

To reduce the variance of the predictions and hedge against in-sample overfitting, we combine the predictions of the best three models listed in the table in the previous section and average the result.

To this end, we define the following `generate_predictions()` function, which receives the model parameters as inputs, loads the weights for the models for the desired epoch, and creates forecasts for the cross-validation and out-of-sample periods (showing only the essentials here to save some space):

```
def generate_predictions(dense_layers, activation, dropout,
                         batch_size, epoch):
    checkpoint_dir = Path('logs')
    checkpoint_path = checkpoint_dir / dense_layers / activation /
                      str(dropout) / str(batch_size)

    for fold, (train_idx, test_idx) in enumerate(cv.split(X_cv)):
        x_train, y_train, x_val, y_val = get_train_valid_data(X_cv, y_cv,
                                                              train_idx,
                                                              test_idx)
        x_val = scaler.fit(x_train).transform(x_val)
        model = make_model(dense_layers, activation, dropout, input_dim)
        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)
```

We store the results for evaluation with Alphalens and a Zipline backtest.

Evaluating signal quality using Alphalens

To gain some insight into the signal content of the ensembled model predictions, we use Alphalens to compute the return differences for investments into five equal-weighted portfolios differentiated by the forecast quantiles (see *Figure 17.9*). The spread between the top and the bottom quintile equals around 8 bps for a one-day holding period, which implies an alpha of 0.094 and a beta of 0.107:

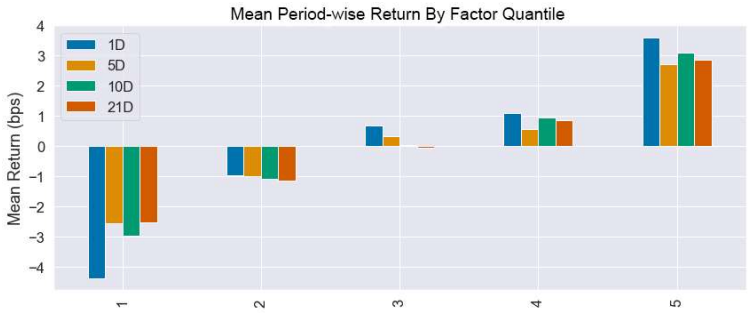

Figure 17.9: Signal quality evaluation

Backtesting the strategy using Zipline

Based on the Alphalens analysis, our strategy will enter long and short positions for the 50 stocks with the highest positive and lowest negative predicted returns, respectively, as long as there are at least 10 options on either side. The strategy trades every day.

The charts in *Figure 17.10* show that the strategy performs well in- and out-of-sample (before transaction costs):

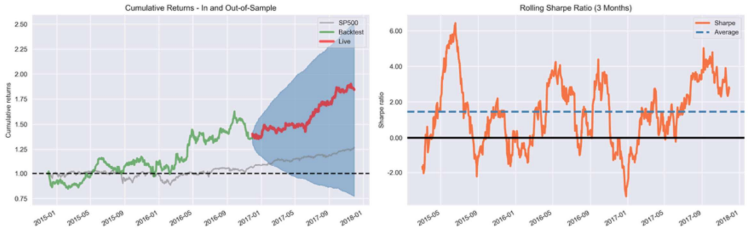

Figure 17.10: In- and out-of-sample backtest performance

It produces annualized returns of 22.8 percent over the 36-month period, 16.5 percent for the 24 in-sample months, and 35.7 percent for the 12 out-of-sample months. The Sharpe ratio is 0.72 in-sample and 2.15 out-of-sample, delivering an alpha of 0.18 (0.29) and a beta of 0.24 (0.16) in/out of sample.

How to further improve the results

The relatively simple architecture yields some promising results. To further improve performance, you can first and foremost add new features and more data to the model.

Alternatively, you can use more sophisticated architectures, including RNNs and CNNs, which are well suited to sequential data, whereas vanilla feedforward NNs are not designed to capture the ordered nature of the features.

We will turn to these specialized architectures in the following chapter.

Summary

In this chapter, we introduced DL as a form of representation learning that extracts hierarchical features from high-dimensional, unstructured data. We saw how to design, train, and regularize feedforward neural networks using NumPy. We demonstrated how to use the popular DL libraries PyTorch and TensorFlow that are suitable for use cases from rapid prototyping to production deployments.

Most importantly, we designed and tuned an NN using TensorFlow and were able to generate tradeable signals that delivered attractive returns

during both the in-sample and out-of-sample periods.

In the next chapter, we will explore CNNs, which are particularly well suited for image data but are also well-suited for sequential data.