Table of Contents

1
Deep Learning

This chapter kicks off part four, which covers several deep learning techniques and how they can be useful for investment and trading. The unprecedented breakthroughs that **deep learning** (**DL**) has achieved in many domains, from image and speech recognition to robotics and intelligent agents, 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.

The enormous DL progress over the last five to ten years builds on ideas that date back decades. However, to realize their potential, these ideas needed to operate at scale, which in turn required complementary advances in the availability of computational resources and large datasets.

In this chapter, we will present feedforward neural networks to introduce central elements of neural network architectures, demonstrate how to efficiently train large models using the backpropagation algorithm, and manage the risks of overfitting. We will also show how to use the popular Keras, TensorFlow 2.0, and PyTorch frameworks, which we will leverage throughout part four.

In the following chapters, we will build on this foundation to design and train a variety of architectures suitable for different investment applications with a particular focus on alternative data sources. 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. We will also cover deep unsupervised learning, including **Generative Adversarial Networks** (**GANs**), to create synthetic data and reinforcement learning to train agents that interactively learn from their environment.

In particular, this chapter will cover the following topics:

- How DL solves AI challenges in complex domains
- How key innovations have propelled DL to its current popularity
- How feedforward networks learn representations from data
- How to design and train deep neural networks in Python
- How to implement deep neural networks using Keras, TensorFlow, and PyTorch
- How to build and tune a deep neural network to predict asset price movement

The code samples and references are in this chapter's directory of the GitHub repository at `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`.

Deep learning and AI

The **machine learning** (**ML**) algorithms covered in part two work well on a wide variety of important problems, including- on text data, as demonstrated in part three. We have also seen how they can provide critical input to a trading strategy. They have been less successful, however, in solving central problems in AI such as recognizing speech or classifying objects in images. The limitations of traditional algorithms to generalize well on such tasks have contributed to the motivation for developing DL, and the numerous breakthroughs by DL have greatly contributed to a resurgence of interest in AI.

In this section, we outline how DL overcomes many of the limitations of other ML algorithms on AI tasks to clarify the assumptions DL makes about data and its relationship with the outcome. These limitations particularly constrain performance on high-dimensional and unstructured data that requires sophisticated efforts to extract informative features.

We also saw that the ML techniques in part two and part three 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 13`, *Working with Text Data*. DL also overcomes the challenge of designing effective features by learning a representation of the data that more efficiently captures its characteristics.

More specifically, we will see how DL is a specific approach to AI and ML that focuses on learning a hierarchical representation of the data, and why this approach works particularly well when applied to high-dimensional, unstructured data, including many popular alternative data sources. We will describe how deep neural networks discover an intricate, hierarchical structure by composing a set of nested functions that compute a new representation in each layer from the representation in the previous layer, and how the backpropagation algorithm adjusts the internal network parameters to enable the learning of these representations from data.

We will also briefly outline how DL broadly fits into the evolution of AI and the diverse set of approaches that aim to achieve the current goals of AI, which focus on solving problems that are easy for humans but difficult to describe using a set of rules, in contrast to tasks that involve straightforward rules such as playing chess.

The challenges of high-dimensional data

We have seen how 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 when we covered the curse of dimensionality in `Chapter 12`, *Unsupervised Learning*.

One aspect of this curse is that volume grows exponentially with the number of dimensions: the volume of a hypercube with edge length 10 increases from 10^3 to 10^4 as the number of dimensions goes from three to four. Consequently, the number of data points required to maintain a given density of observations also grows exponentially.

Moreover, functional relationships may become more complex when they can exhibit distinct behavior in each of a larger number of dimensions. Traditional ML techniques struggle to learn and generalize complicated functions in high-dimensional spaces. We explained in `Chapter 6`, *The Machine Learning Process*, that many ML problems are intractable if we expect the algorithm to learn functions that vary arbitrarily across all dimensions, simply because there are too many candidates. Instead, algorithms hypothesize that the target function belongs to a certain class to impose constraints that enable a successful search for the optimal solution to the prediction 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, most clearly illustrated by the k-nearest neighbor algorithm (see `Chapter 6`, *The Machine Learning Process*).

In high-dimensional space, the data density drops exponentially, so that unless we add a corresponding number of observations, the training samples will be further and further apart. Hence, the notion of nearby training examples 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 ordinary tasks 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, we saw that how much value an ML algorithm adds to a trading strategy depends to a great extent on our ability to engineer features that represent the predictive information in the source data so that the algorithm can process it. Ideally, the representation captures independent drivers of the outcome, as we discussed in `Chapter 4`, *Alpha Factor Research*, and throughout part two and part three when investigating factors that capture trading signals.

Representation learning allows an ML algorithm to automatically discover the representation of the raw data that is most useful for detecting or classifying patterns. DL combines this technique with specific assumptions about the nature of the features.

How DL extracts hierarchical features from data

The core idea behind DL is that a composition of factors, or features, potentially organized in a hierarchy of multiple levels, has generated the data. Hence, a deep model encodes the prior belief that the target function is composed of simpler functions. This assumption allows 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 using neural network architectures that compose simple but non-linear functions and successively transform the representation from one level (starting with the raw input) into a new representation at a higher, slightly more abstract level. These successive transformations can also be interpreted as learning a computer program that takes multiple, sequential steps. By combining enough of these transformations, DL is able to learn very complex functions.

Applied to a classification task, for example, higher representation levels tend to amplify the aspects of the data most helpful for discriminating different objects while suppressing irrelevant sources of variation. As we will see in more detail in `Chapter 18`, *Convolutional Neural Networks*, raw image data is just a two-or three-dimensional array of pixel values. The first representation layer 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.

Consequently, it is no surprise that the rise of DL parallels the large-scale availability of unstructured data. The superior ability of DL to model the types of data associated with AI tasks has enabled a variety of new use cases. To the extent that these data sources also figure prominently among alternative data, DL has become similarly relevant for algorithmic trading.

Universal function approximation

The Universal Approximation Theorem formalizes the powerful ability of neural networks to capture arbitrary relationships between input and output data. In 1989, George Cybenko showed that neural networks with a single layer of neurons connecting input and output using nonlinear, sigmoid activation functions are generally able to represent any continuous function on a closed and bounded subset of R^n.

Kurt Hornik showed in 1991 that it is not the specific shape of the activation function but rather the multi-layered architecture that enables the hierarchical feature representation that allows neural networks to approximate universal functions.

However, the theorem also does not specify the network architecture required to actually represent the target function. We will see later in the section on neural network architecture that there are numerous choices, from the width and depth of the networks to the connections between neurons and the type of activation functions to use.

Furthermore, the ability to represent arbitrary functions does not imply that a network can actually learn the parameter's function. It took over two decades for backpropagation, the most popular learning algorithm for neural networks today, which had been discovered independently in different contexts by 1986, to become effective at scale, as we will see in the corresponding section later in this chapter.

DL and manifold learning

There is an important conceptual reason why DL works is the manifold hypothesis, which we encountered in `Chapter 12`, *Unsupervised Learning*. The idea is that high-dimensional phenomena can often be represented well in lower dimensions, and if we can find this representation, then we can reduce or even avoid the challenges posed by the curse of dimensionality.

A manifold refers to a connected set of points, typically in high-dimensional space, that can be approximated well using only a much smaller number of dimensions. In other words, the lower-dimensional manifold is embedded in a higher-dimensional space. The example of a street as a one-dimensional manifold in a three-dimensional space illustrates how house numbers are much simpler descriptors than three-dimensional coordinates.

There are strong arguments to support the manifold hypothesis. Out of the very large number of potential images, text strings, or sounds, only a very small number is meaningful. In other words, the probability distribution over these data sources is highly concentrated on specific feature configurations rather than uniform.

Moreover, there are often small variations of the input that maintain the validity of the input. For image data, for instance, adjustments to brightness, luminance, changes to colors, rotations, and so on could be considered movements on the manifold for a given object.

The hierarchical representations learned by DL may approximate the underlying manifolds, and we will see in more detail in `Chapter 19`, *Unsupervised Deep Learning*, how autoencoders accomplish this.

How DL relates to ML and AI

The current level of public attention and debate warrants a brief outline of how DL relates to AI and ML (see references on GitHub for more detail: `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`).

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. ML is an important subfield that also has a long history in related disciplines, such as statistics, and became prominent in the 1980s. As discussed in the previous section, DL is a form of representation learning, itself a subfield of ML.

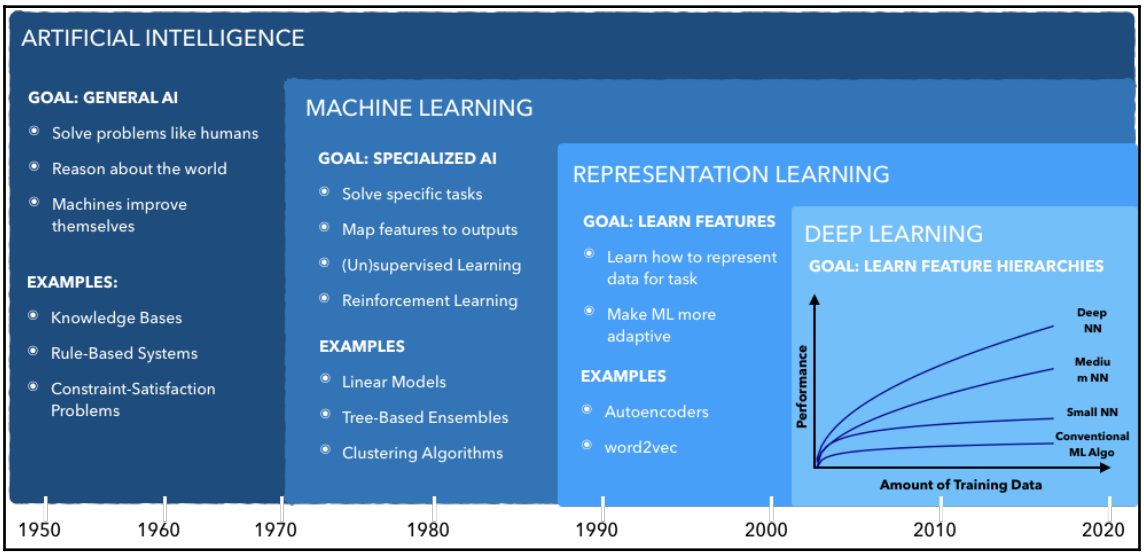
The initial goal of AI was to achieve General AI, conceived as the ability to solve problems considered to require human-level intelligence, and to reason and draw logical conclusions about the world and automatically improve itself. AI applications that do not involve ML include knowledge bases that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

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 and includes most of the material covered in the book so far.

The previous section introduced representation learning as a technique to automatically extract features from data that are suitable for a given ML task. We covered fundamental techniques such as **Principal Component Analysis** (**PCA**) and **Independent Component Analysis** (**ICA**) in `Chapter 12`, *Unsupervised Learning*. Another example is the Word2vec word embedding algorithm, which we introduced in `Chapter 15`, *Word Embeddings*.

Finally, DL extracts a hierarchical representation that is often capable of improving its performance by adding layers as the training data grows. The composition of multiple processing layers that learn multiple levels of abstraction to represent the data has dramatically improved the state of the art in speech recognition, visual object recognition, and detection, and many other domains, such as drug discovery and genomics.

The following figures shows these relationships and emphasizes the key strength of DL: the ability to improve its predictive performance by adding more layers as more data becomes available:



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

How to design a neural network

DL relies on neural networks, which consist of a few key building blocks, which in turn can be configured in a multitude of ways. In this section, we will introduce how neural networks work and illustrate the most important components used to design different architectures, including types of hidden and output units, cost functions, and various options to connect these components.

Neural networks, also called *artificial neural networks*, were inspired by biological models of learning as represented by 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 neural network 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: while the number of neurons used in neural network has more than doubled since the 1950s, they will only reach the scale of the human brain around 2050.

We will also explain how backpropagation uses gradient information to adjust all neural network parameters based on training errors. The composition of various non-linear modules implies that the optimization of the objective function using backpropagation can be quite challenging. We also introduce various algorithms that aim to accelerate the learning process.

How neural networks work

In this section, we will introduce a fundamental class of (artificial) neural networks that is based on the **Multilayer Perceptron** (**MLP**) and consists of one or more hidden layers that connect the input to the output layer.

This class of neural networks is also called **feedforward neural networks** because information only flows in one direction, from input to output. Hence, they can be represented as directed acyclic graphs. In contrast, in the next chapter, we cover RNNs, which include loops from the output back to the input to enable the neural network to keep track of or memorize past patterns and events.

We will first describe the architecture and how to implement it using NumPy. Then we will describe backpropagation and implement this learning algorithm in Python to train a simple network and demonstrate how it is capable of binary classification where the classes are not linearly separable. See the `build_and_train_feedforward_nn` notebook for implementation details.

A simple feedforward network architecture

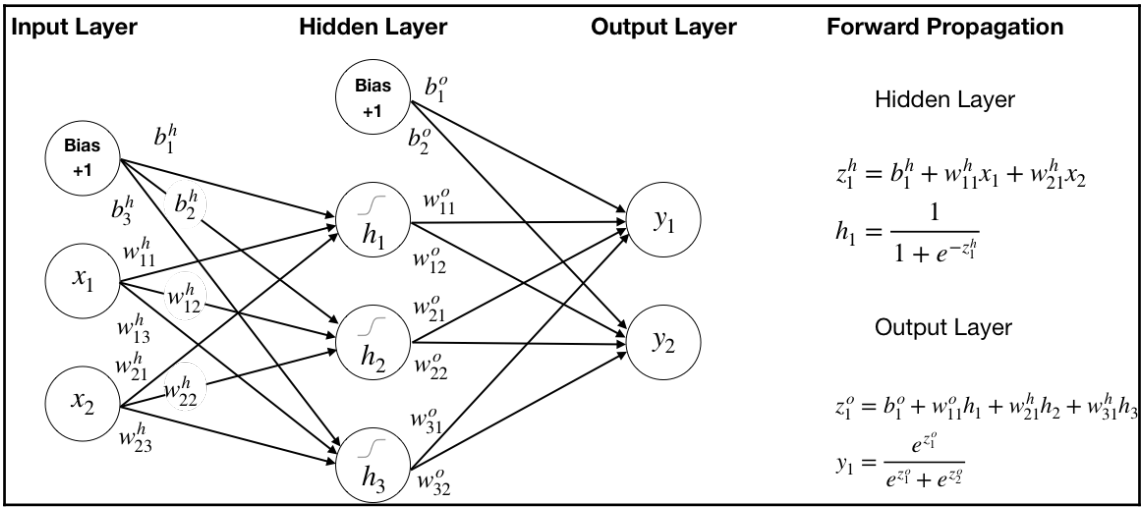
A feedforward neural network consists of several layers, each of which receives input data and produces an output. The chain of transformations starts with the input layer, that passes the source data to one or several internal or hidden layers, and the output layer that computes a result that can be compared to the outcome or label of the ML problem.

The hidden and output layers consist of nodes or neurons, each of which connects to some or all nodes of the previous layer, where the latter is called a **fully-connected** or **dense layer**. The network architecture can be summarized by the depth of the neural network, measured by the number of hidden layers (hence *deep* learning), and the width, or the number of nodes of each layer.

Each connection has a weight, which is used to compute a linear combination of the input values. A layer may also have a bias node without any inputs that always outputs a *1* and is used by the nodes in the subsequent layers like a constant in linear regression. The result is called an **affine transformation**. The weights and biases are the parameters that determine the network output and predictive performance, and learning optimal values for these parameters is the goal of the training phase.

The hidden layers usually also perform a non-linear transformation of the result, for example, based on the sigmoid function used for logistic regression (see `Chapter 7`, *Linear Models*) to produce an activation that becomes the input to the subsequent layer. The output layer will process the linear combination of the inputs from the last hidden layer according to the type of ML problem, such as regression, binary, or multi-class classification.

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



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 diagram 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 inputs, so that it performs a logistic regression with output h_1. Hence, the hidden layer runs three logistic regressions in parallel that generally will produce different parameters or weights as 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 functions to multiple classes and squashes the hidden layer values so they represent probabilities for the classes (only two in this case); that is, it forces the outputs to be non-negative and sum to 1. We could represent the output using a single binary variable, but neural networks are often used for multi-class problems and this representation allows us to demonstrate the softmax output function.

Forward propagation computed by chained non-linear transformations can also be expressed as nested functions, where `h` again represents the hidden layer and `o` represents the output layer, to produce the neural network estimate of the output, as follows:

$$\hat{y} = o(h(x))$$

Key design choices

The design choices for neural networks are similar to other supervised learning models in some regards. For example, the neural network model output is usually dictated by the nature of the training labels and the type of ML problem they represent, such as regression, classification, or ranking. Based on the output, we need to select a cost function and an optimization algorithm to minimize this objective represented by the cost function.

Neural network-specific choices include the overall architecture, that is, the neural network's depth in terms of number of layers, and their relspective widths or number of nodes, as well as the design of connections between nodes of different layers. A key concern is how efficiently the backpropagation algorithm translates training errors into adequate parameter adjustments based on the information provided by the gradient. The functional forms of nonlinear elements in hidden and output layers, for example, can facilitate or hinder the flow of this information. 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 reduce the network's capacity and possibly reduce the generalization error, while also reducing the computational cost. Finally, each hidden layer requires a choice of the activation function.

Cost functions

The cost functions for neural networks do not differ significantly from those for other models. The choice can, however, impact the ability to train the model because it interacts with the nature of the gradient of the output.

Historically, the **mean squared error** (**MSE**) was a common choice, but slowed down training with binary sigmoid or multi-class softmax outputs. The gradients for these output functions can be very low (for example, for the flat region of the sigmoid function, called **saturation**) so that backpropagation can take a long time to achieve significant parameter updates, which in turn slows down training. The use of the cross-entropy family of loss functions greatly improved the performance of these models by reducing saturation.

Output units

Neural networks are applied to common supervised learning problems and, hence, use familiar output representations of the final hidden layer activations:

- Linear output units compute an affine transformation from the hidden layer activations and are common for regression problems in conjunction with MSE cost.
- Sigmoid output units model a Bernoulli distribution, just like logistic regression, with hidden activations as input.
- Softmax units generalize the logistic sigmoid and model a discrete distribution over more than two classes as demonstrated precedingly.

Hidden units

Hidden units are unique to the design of neural networks, and several non-linear activation functions have been used successfully. The design of hidden activation functions remains an area of research because it has a critical impact on the training process.

A very popular class of activation functions are piece-wise linear units, such as the **Rectified Linear Unit** (**ReLU**). The functional form is similar to the payoff for a call option and the activation is computed as $g(z) = max(0, z)$ for a given activation, z. As a result, the derivative is constant whenever the unit is active. ReLUs are usually combined with an affine transformation of the inputs. They are often used instead of sigmoid units and their discovery has greatly improved the performance of feedforward networks. They are often recommended as the default.

There are several ReLU extensions that aim to address the limitations of ReLU to learn using gradient descent when they are not active and their gradient is zero. See the references on GitHub for details at `https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`.

An alternative to the logistic sigmoid activation function, σ, is the hyperbolic tangent, 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 identify function, so that for small activation values, the network behaves more like a linear model, which in turn facilitates training.

How to regularize deep neural networks

The large capacity of neural networks to approximate arbitrary functions greatly increases the risk of overfitting. We have seen for all models so far that 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 constraints used with decision trees and tree-based ensemble models.

Frequently, regularization takes the form of a soft constraint on parameter values that trades off some additional bias for lower variance. Sometimes the constraints and penalties are designed to encode prior knowledge. Other times, these constraints and penalties are designed to express a general preference for a simpler model class. 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.

The best protection against overfitting is to train the model on a larger dataset. Data augmentation, for example, by creating slightly modified versions of images, is a powerful alternative approach. Popular regularization techniques for neural networks that we will apply throughout part four, and that can also be used in combination, include parameter norm penalties, early stopping, and dropout.

Parameter norm penalties

We have encountered parameter norm penalties as L1 regularization and the corresponding lasso regression as L2 regularization and Ridge regression in `Chapter 7`, *Linear Models*.

In the context of DL, 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 neural networks, the bias parameters are usually not constrained, only the weights. Sometimes different penalties or hyperparameter values are used for different layers, but the added tuning complexity quickly becomes prohibitive.

L2 regularization preserves directions along which the parameters contribute significantly to reduce the objective function. L1 regularization, in contrast, has the ability to produce sparse parameter estimates by reducing weights all the way to zero. We have seen how lasso regression can be used for linear feature selection as a result.

Early stopping

We encountered early stopping as a regularization technique in `Chapter 10`, *Decision Trees and Random Forests*. It is probably the most commonly used form of regularization in DL and is popular because it is both effective and simple to use.

It works as a regularization mechanism by monitoring the model's performance on a validation set during training. When the performance ceases to improve for a certain number of observations, the algorithm stops to prevent overfitting.

Early stopping can be viewed as a very efficient hyperparameter selection algorithm that automatically determines the correct amount of regularization, whereas parameter penalties require hyperparameter tuning to identify the ideal weight decay.

Dropout

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

Dropout is a very popular regularization technique because it is computationally very inexpensive and does not significantly constrain the type of model or training procedure that can be used. A downside is that more iterations are necessary to achieve the same amount of learning, but on the plus side, each iteration is faster due to lower computational cost.

It works as a regularization technique because it prevents units from co-adapting to or compensating for mistakes made by other units during the learning process, thereby increasing the risk of overfitting. One of the key insights of dropout is that training a network with stochastic behavior and making predictions by averaging over multiple stochastic decisions implements a form of bagging with parameter sharing.

Optimization for DL

Training a deep neural network is very challenging and time-consuming due to the non-convex objective function. 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, which can result from multiplying several large weights, can cause excessive adjustments.
- Deep architectures or the modeling of long-term dependencies in RNNs (see the next chapter) can require the multiplication of many weights, leading to vanishing gradients, so that at least parts of the neural network receive few or no updates.

Several algorithms have been developed to address some of these challenges, such as variations of **Stochastic Gradient Descent** (**SGD**) and approaches that use adaptive learning rates. There is no single best algorithm, though, although adaptive learning rates have shown some promise.

SGD

Gradient descent iteratively adjusts the neural network parameter using the information by the gradient. 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, as follows:

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

The gradient can be evaluated for all training data, a randomized batch of data, or individual observations (called *online learning*). Random samples imply SGD, which often leads to faster convergence if random samples provide an unbiased estimate of the gradient direction at any point in the iterative process.

There are numerous challenges because the objective function is not convex. It can be difficult to define a learning rate or a rate schedule ex-ante that facilitates efficient convergence—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.

While, in practice, the dimensionality would be much higher, the image of a local optimum at the center of an elongated ravine is often used. It implies a minimum inside a deep and narrow gorge or canyon with very steep walls that have a large gradient but a much gentler slope toward a local minimum at the bottom of this region. Gradient descent naturally follows the steep gradient and will make repeated adjustments up and downs the walls of the canyons with much slower movements toward the minimum.

Momentum aims to address such situation by tracking past 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, as follows:

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

Adaptive learning rates

The choice of appropriate learning rates is very challenging as highlighted in the preceding subsection on SGD. At the same time, it is one of the most important parameters and 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 the momentum rate, another hyperparameter.

Several algorithms aim to adapt the learning rate throughout the training process based on gradient information (see references on GitHb for more detail).

AdaGrad

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

AdaGrad is designed to perform well on convex functions and has had mixed performance in a DL contexts because it can reduce the learning rate too fast 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 use in practice.

Adam

Adam stands for Adaptive Moments. It combines aspects of RMSProp with momentum. It is considered fairly robust and is often used as default.

How to build a neural network using Python

To gain a better understanding of how neural networks work, we will formulate the preceding architecture and forward propagation computations using matrix algebra and implement it using NumPy, the Python counterpart of linear algebra.

The input layer

The preceding architecture is designed for two-dimensional input data, `X`, which represent two different classes, `Y`. In matrix form, both `X` and `Y` are of shape N x 2, as follows:

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

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

```
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, as follows:

```
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,)'
```

See the notebook for a visualization of the result.

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 x 3 matrix, W^h, and the hidden layer bias vector by a three-dimensional vector, as follows:

$$\underset{2\times3}{\mathbf{W^h}} = \begin{bmatrix} w^h_{11} & w^h_{12} & w^h_{13} \\ w^h_{21} & w^h_{22} & w^h_{23} \end{bmatrix} \qquad \underset{1\times3}{\mathbf{b^h}} = \begin{bmatrix} b^h_1 & b^h_2 & b^h_3 \end{bmatrix}$$

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, as follows:

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

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

```
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, as follows:

```
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 x 2 weight matrix, W^o, and a two-dimensional bias vector, b^o, as follows:

$$\underset{3\times 2}{\mathbf{W}^o} = \begin{bmatrix} w_{11}^o & w_{12}^o \\ w_{21}^o & w_{22}^o \\ w_{31}^o & w_{32}^o \end{bmatrix} \qquad \underset{1\times 2}{\mathbf{b}^o} = \begin{bmatrix} b_1^o & b_2^o \end{bmatrix}$$

The linear combination of the hidden layer outputs results in an N x 2 matrix, Z^o, as follows:

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

The output layer activations are computed by the softmax function, ς, which normalizes the Z^o to conform to the conventions used for discrete probability distributions, as follows:

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

We define 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 earlier, the output layer activations depend on the hidden layer activations and the output layer weights and biases, as follows:

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

Now we have all the components to integrate the layers and compute the neural network 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, as follows:

```
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, as follows:

```
def predict(data, hidden_weights, hidden_bias, output_weights,
output_bias):
    """Predicts class 0 or 1"""
    y_pred_proba = forward_prop(data,
        q                          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 neural network 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, as follows:

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

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 train a neural network

The goal of neural network training is to adjust the hidden and output layer parameters to best predict new data based on training samples. Backpropagation, often simply called **backprop**, ensures that the information about the performance of the current parameter values gleaned from the evaluation of the cost function for one or several samples flows back to parameters and facilitates optimal updates.

Backpropagation refers to the computation of the gradient of the function that relates the internal parameters that we wish to update to the cost function. The gradient is useful because it indicates the direction of parameter change, which causes the maximal increase in the cost function. Hence, adjusting the parameters in the direction of the negative gradient should produce an optimal cost reduction for the observed samples, as we saw in `Chapter 6`, *Linear Models*.

However, gradient descent optimization algorithms do not offer guarantees of convergence when applied to non-convex functions such as neural network gradients with non-linear activation and cost functions. The gradient is only valid for an infinitely small step, and a key challenge for the optimization algorithm is to define a discrete step size that measurably changes the parameters while avoiding an adverse change in the cost function.

We usually define this step size as the learning rate. Moreover, the optimization result is sensitive to the starting values, which has given rise to various weight initialization strategies.

The next section on design choices introduces several optimization algorithms tailored to non-convex optimization, including approaches that automatically adapt the learning rate.

How to implement backprop using Python

To update the neural network 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 neural network comprises a set of nested functions as highlighted precedingly. 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 $z \in R^m$ and $x \in R^n$ so that the hidden layer, h, maps from R^n to R^m and $z = h(x)$ and $y = o(z)$, we get the following:

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

We can express this more concisely using matrix notation using the `m` x `n` Jacobian matrix of `h`, as follows:

$$\frac{\partial z}{\partial x}_{m \times n}$$

That 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` that contains all partial derivatives can thus be written as follows:

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

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 notebook for details), on the left for scalar values and on the right in matrix notation, as follows:

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

We define the `loss_gradient` function accordingly, as follows:

```
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 the `J`, with respect to the weight matrix, as follows:

$$\frac{\partial J}{\partial \mathbf{W}^o} = \frac{\partial J}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{Z}^o} \frac{\partial \mathbf{Z}^o}{\partial \mathbf{W}^o} = \frac{\partial J}{\partial \mathbf{Z}^o} \frac{\partial \mathbf{Z}^o}{\partial W^o} = \mathbf{H}^T (\hat{\mathbf{Y}} - \mathbf{Y}) = \mathbf{H}^T \delta^o$$

We can now define `output_weight_gradient` and `output_bias_gradient` accordingly, both taking the loss gradient, δ^o, 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 º refers to the element-wise matrix product:

$$\nabla_{\mathbf{Z}^h} J = \mathbf{H} \circ (1 - \mathbf{H}) \circ \left[\delta_o \cdot \left(\mathbf{W}^{\mathbf{o}}\right)^T\right] = \delta_h$$

We define a `hidden_layer_gradient` function to encode this result, as follows:

```
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 as follows:

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

The corresponding functions are as follows:

```
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 preceding gradient definition and computes the relevant weight and bias updates from the training data and labels, and the current weight and bias values, as follows:

```
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 numerical to the analytical gradient derived precedingly. It does so by slightly perturbing individual parameters and validates that the 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 preceding `compute_gradients` function with the most recent momentum updates as follows for each parameter matrix:

```
def update_momentum(X, y_true, param_list, Ms, momentum_term, eta):
    """Compute updates with momentum."""
    gradients = compute_gradients(X, y_true, *param_list)
    return [momentum_term * momentum - eta * 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

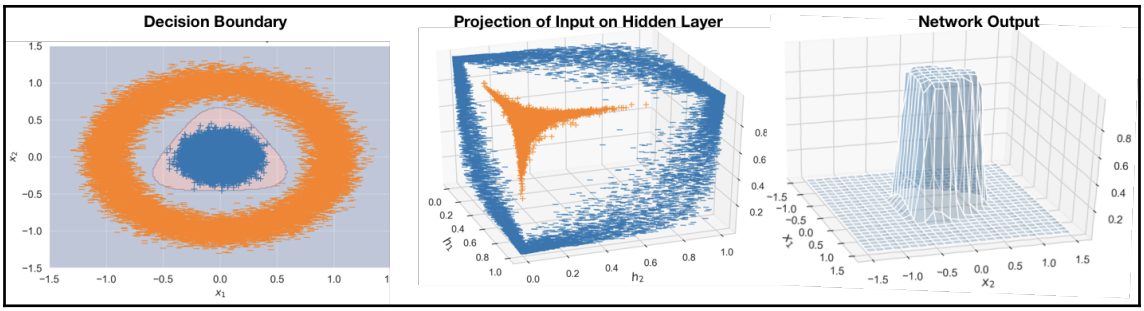
To train the network, we first randomly initialize all network parameters using a standard normal distribution (see notebook). For a given number of iterations or epochs, we run momentum updates and compute the training loss, as follows:

```
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
```

The notebook 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 quickly. We have not used SGD, which would have likely significantly accelerated convergence.

The following plots 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, as shown on the left. The decision boundary misclassifies very few data points and would further improve with continued training.

The center plot shows the representation of the input data learned by the hidden layer. The network learns hidden layer 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:



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 non-linear classification problem using backprop and gradient descent with momentum.

We will next review key design choices useful to design and train more complex architectures before we turn to popular DL libraries that facilitate the process by providing many of these building blocks and automating the differentiation process to compute the gradients and implement backpropagation.

How to use DL 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 just having released version 1.0 and TensorFlow 2.0 expected in early Spring 2019, when it is expected to adopt Keras as its main interface.

All libraries provide the building blocks we discussed previously under *Design choices, regularization and optimization algorithms*, and facilitate fast training on **Graphics Processing Units** (**GPUs**). The libraries differ a bit in their focus with TensorFlow, which was originally designed for deployment in production, and Keras, which is more tailored for fast prototyping, although the interfaces are gradually converging.

We will illustrate the use of these libraries using the same network architecture and dataset as in the previous example.

How to use Keras

Keras was designed as a high-level or meta API to accelerate the iterative workflow when designing and training deep neural networks with computational backends, such as TensorFlow, Theano, or CNTK. It has been integrated into TensorFlow in 2017 and is set to become the principal TensorFlow interface with the 2.0 release. You can also combine code from both libraries to leverage Keras' high-level abstractions as well as customized TensorFlow graph operations.

Keras supports both a slightly simpler sequential and 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 using 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 following `fit` method.

Keras infers the sizes of the inputs received by other layers from the previous layer's units argument, as follows:

```
from keras.models import Sequential
from 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'),
])
```

Keras 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 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:

```
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 next section), as follows:

```
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, as follows:

```
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 manual network implementation. The training with Keras runs a multiple faster, though.

How to use TensorBoard

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

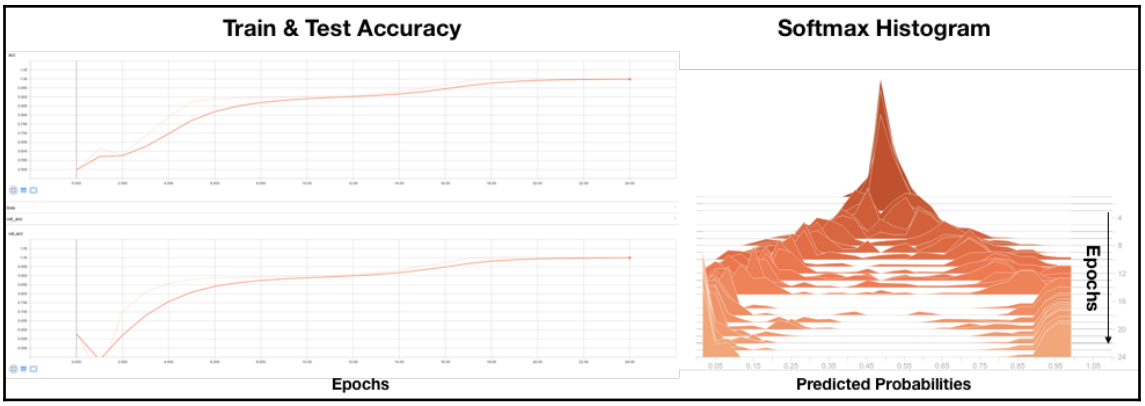
You can use it to visualize the computational graph, plot various execution and performance metrics, and even visualize image data processed by the network. It also permits comparisons of different training runs.

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

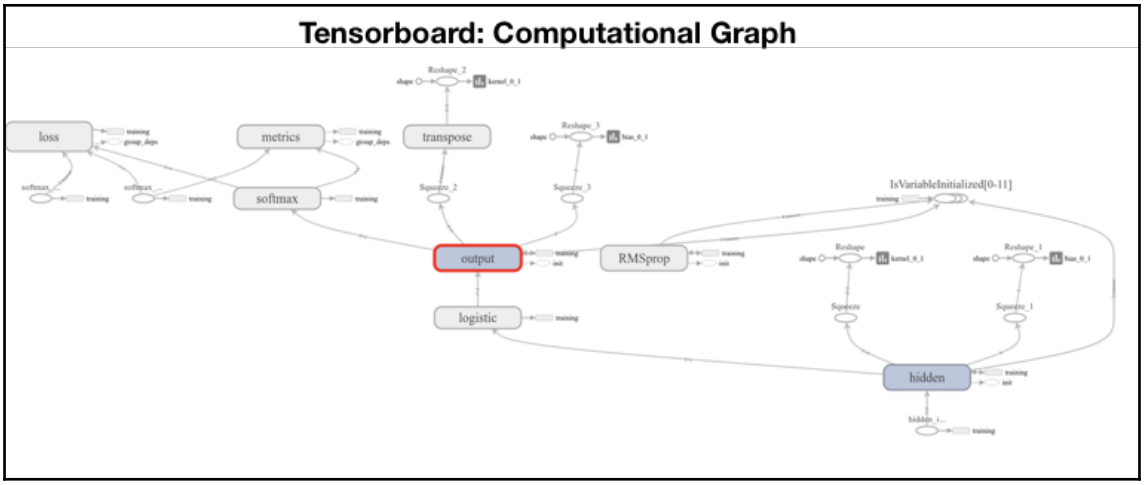
```
tensorboard --logdir=/full_path_to_your_logs ## e.g. ./tensorboard
```

For starters, the visualizations include train and validation metrics (see the left panel of the following diagram).

In addition, you can view histograms of the weights and biases over various epochs (in the right panel, 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:



Computational graphs can become fairly complicated with thousands or millions of parameters. The visualization for our simple example architecture already includes numerous components. These visualizations are very useful when debugging, and Keras and TensorFlow offer numerous tools to organize your network using named scopes. See the links to more detailed tutorials on GitHub (`https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading`) for further reference:



How to use PyTorch 1.0

PyTorch has been developed at the Facebook AI research group led by Yann LeCunn and the first alpha version was released in September 2016. It provides deep integration with Python libraries such as 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, for example, 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 (`https://pytorch.org/docs/stable/index.html`).

The resulting flexibility combined with an intuitive Python-first interface and speed of execution have 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` instance 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 a neural network 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, as follows:

```
import torch.nn as nn

class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()                    # Inherited from the
parent class 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, as follows:

```
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 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, as follows:

```
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`, as follows:

```
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, as follows:

```
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.

How to use TensorFlow 2.0

TensorFlow has become the leading deep learning library shortly after its release in September 2015, one year before PyTorch. TensorFlow 2.0 aims to simplify the API that has grown increasingly complex over time by making the Keras API, integrated into TensorFlow as part of the `contrib` package since 2017, its principal interface, and adopting eager execution. It will continue to focus on a robust implementation across numerous platforms but will make it easier to experiment and do research.

The `how_to_use_tensorflow` notebook illustrates how to use the 2.0 release (updated as the interface stabilizes).

How to optimize neural network architectures

In practice, we need to explore variations of the design options outlined previously because we can rarely be sure from the outset of which network architecture best suits the data.

The `GridSearchCV` class provided by scikit-learn that we encountered in `Chapter 6`, *The Machine Learning Process*, conveniently automates this process. Just be mindful of the risk of false discoveries and keep track of how many experiments you are running to adjust the results accordingly.

In this section, we will explore various options to build a simple feedforward neural network to predict asset price movement for a one-month horizon. See the `how_to_optimize_a_NN_architecure` notebook for details.

Creating a stock return series to predict asset price movement

We will use the last 24 monthly returns and dummy variables for the month and the year to predict whether the price will go up or down the following month. We use the daily Quandl stock price dataset (see GitHub for instructions on how to source the data). Run the following code:

```
prices = (pd.read_hdf('../data/assets.h5', 'quandl/wiki/prices')
        .adj_close
        .unstack().loc['2007':])

DatetimeIndex: 4706 entries, 2000-01-03 to 2018-03-27
Columns: 3199 entries, A to ZUMZ
dtypes: float64(3199)
```

We will work with monthly returns to keep the size of the dataset manageable and remove some of the noise contained in daily returns, which leaves us with almost 2,500 stocks with 120 monthly returns each, as follows:

```
returns = (prices
           .resample('M')
           .last()
           .pct_change()
           .loc['2008': '2017']
           .dropna(axis=1)
           .sort_index(ascending=False))
```

```
returns.info()
DatetimeIndex: 120 entries, 2017-12-31 to 2008-01-31
Freq: -1M
Columns: 2489 entries, A to ZUMZ
```

In the next step, we will iteratively select a rolling window of `T=25` consecutive returns for each stock, and transpose the selection so that each row contains the returns for a single stock. We binarize the latest return, depending on whether it was positive or negative, to use it as the outcome label. Then we concatenate the result after appending and one-hot encoding information about the month and year of the target return to obtain over 235,000 return series, as follows:

```
data = pd.DataFrame()
for i in range(n-T-1):
    df = returns.iloc[i:i+T+1]
    data = pd.concat([data, (df.reset_index(drop=True).T
                             .assign(year=df.index[0].year,
                                     month=df.index[0].month))],
                     ignore_index=True)
data[tcols] = (data[tcols].apply(lambda x: x.clip(lower=x.quantile(.01),
                                                  upper=x.quantile(.99))))
data['label'] = (data[0] > 0).astype(int)
data['date'] = pd.to_datetime(data.assign(day=1)[['year', 'month', 'day']])
data = pd.get_dummies((data.drop(0, axis=1)
                       .set_index('date')
                       .apply(pd.to_numeric)),
                      columns=['year', 'month']).sort_index()
data.shape
(236455, 45)
```

Defining a neural network architecture with placeholders

Keras contains a wrapper that we can use with the sklearn `GridSearchCV` class. It requires a `build_fn` instance, which constructs and compiles the model based on arguments that can later be passed during the `GridSearchCV` 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, as follows:

```
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=input_dim))
            model.add(Activation(activation))
        else:
            model.add(Dense(layer_size))
            model.add(Activation(activation))
    model.add(Dropout(dropout))
    model.add(Dense(1))
    model.add(Activation('sigmoid'))

    model.compile(loss='binary_crossentropy',
                  optimizer='Adam',
                  metrics=['binary_accuracy', auc_roc])
    return model
```

Defining a custom loss metric for early stopping

For binary classification, **Area Under the Curve** (**AUC**) is an excellent metric but is not provided by Keras. However, we can define a custom loss metric for use with the early stopping callback as follows (included in the preceding compile step):

```
def auc_roc(y_true, y_pred):
    # any tensorflow metric
    value, update_op = tf.metrics.auc(y_true, y_pred)

    # find all variables created for this metric
    metric_vars = [i for i in tf.local_variables() if 'auc_roc' in
i.name.split('/')[1]]

    # Add metric variables to GLOBAL_VARIABLES collection.
    # They will be initialized for new session.
    for v in metric_vars:
```

```
            tf.add_to_collection(tf.GraphKeys.GLOBAL_VARIABLES, v)

        # force to update metric values
        with tf.control_dependencies([update_op]):
            value = tf.identity(value)
        return value
```

Running GridSearchCV to tune the neural network architecture

We split the data into a training set for cross-validation and a holdout test set using stratified sampling, as the classes are slightly unbalanced, as follows:

```
X_train, X_test, y_train, y_test = train_test_split(features, label,
                                                    test_size=.1,
                                                    random_state=42,
                                                    shuffle=True,
                                                    stratify=data.label)
```

Now we just need to define our Keras classifier using the `make_model` function, set stratified cross-validation, and define the parameters that we would like to explore, as follows:

```
clf = KerasClassifier(make_model, epochs=10, batch_size=32)
cv = StratifiedKFold(n_splits=5, shuffle=True)
param_grid = {'dense_layers': [[64], [64, 64], [96, 96], [128, 128]],
              'optimizer': ['RMSprop', 'Adam'],
              'activation': ['relu', 'tanh'],
              'dropout': [.25, .5, .75]}
```

To trigger the parameter search, we instantiate a `GridSearchCV` object, define the `fit_params` that will be passed to the Keras model's `fit` method, and provide the training data to the `GridSearchCV` fit method, as follows:

```
gs = GridSearchCV(estimator=clf,
                  param_grid=param_grid,
                  scoring='roc_auc',
                  cv=cv)
fit_params = dict(callbacks=[EarlyStopping(monitor='auc_roc',
                                           patience=300,
                                           verbose=1, mode='max')],
                                           verbose=2,
                                           epochs=50)

gs.fit(X=X_train.astype(float), y=y_train, **fit_params)
```

```
gs.best_estimator_.model.save('best_model.h5')
```

The result shows that the network achieves an AUC score of 0.77 using two-layers with `64` units each, rectified linear activation functions, and a dropout rate of `0.5`:

```
print('\nBest Score: {:.2%}'.format(gs.best_score_))
print('Best Params:\n', pd.Series(gs.best_params_))

Best Score: 77.39%
Best Params:
     activation relu
     dense_layers [64, 64]
     dropout 0.5
```

How to further improve the results

The relatively simple architecture yields some promising results. To further improve performance, you can do the following:

- First and foremost, add new features and more data to the model
- Expand the set of architectures to explore, including more or wider layers
- Inspect the training progress and train for more epochs if the validation error continued to improve at 50 epochs

Finally, you can use more sophisticated architectures, including RNNs and CNNs, that are well suited to sequential data, whereas vanilla feedforward neural networks 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 Keras, PyTorch, and TensorFlow, which are suitable for use cases from rapid prototyping to production deployments.

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

Index

P

R

S

T