11

Implementing a Multilayer Artificial Neural Network from Scratch

As you may know, deep learning is getting a lot of attention from the press and is, without doubt, the hottest topic in the machine learning field. Deep learning can be understood as a subfield of machine learning that is concerned with training artificial **neural networks** (**NNs**) with many layers efficiently. In this chapter, you will learn the basic concepts of artificial NNs so that you are well equipped for the following chapters, which will introduce advanced Python-based deep learning libraries and **deep neural network** (**DNN**) architectures that are particularly well suited for image and text analyses.

The topics that we will cover in this chapter are as follows:

- Gaining a conceptual understanding of multilayer NNs
- Implementing the fundamental backpropagation algorithm for NN training from scratch
- Training a basic multilayer NN for image classification

Modeling complex functions with artificial neural networks

At the beginning of this book, we started our journey through machine learning algorithms with artificial neurons in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*. Artificial neurons represent the building blocks of the multilayer artificial NNs that we will discuss in this chapter.

The basic concept behind artificial NNs was built upon hypotheses and models of how the human brain works to solve complex problem tasks. Although artificial NNs have gained a lot of popularity in recent years, early studies of NNs go back to the 1940s, when Warren McCulloch and Walter Pitts first described how neurons could work. (*A logical calculus of the ideas immanent in nervous activity*, by *W. S. McCulloch* and *W. Pitts*, *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.)

However, in the decades that followed the first implementation of the **McCulloch-Pitts neuron** model—Rosenblatt's perceptron in the 1950s—many researchers and machine learning practitioners slowly began to lose interest in NNs since no one had a good solution for training an NN with multiple layers. Eventually, interest in NNs was rekindled in 1986 when D.E. Rumelhart, G.E. Hinton, and R.J. Williams were involved in the (re)discovery and popularization of the backpropagation algorithm to train NNs more efficiently, which we will discuss in more detail later in this chapter (*Learning representations by backpropagating errors*, by *D.E. Rumelhart*, *G.E. Hinton*, and *R.J. Williams*, *Nature*, 323 (6088): 533–536, 1986). Readers who are interested in the history of **artificial intelligence** (**AI**), machine learning, and NNs are also encouraged to read the Wikipedia article on the so-called *AI winters*, which are the periods of time where a large portion of the research community lost interest in the study of NNs (`https://en.wikipedia.org/wiki/AI_winter`).

However, NNs are more popular today than ever thanks to the many breakthroughs that have been made in the previous decade, which resulted in what we now call deep learning algorithms and architectures—NNs that are composed of many layers. NNs are a hot topic not only in academic research but also in big technology companies, such as Facebook, Microsoft, Amazon, Uber, Google, and many more that invest heavily in artificial NNs and deep learning research.

As of today, complex NNs powered by deep learning algorithms are considered state-of-the-art solutions for complex problem solving such as image and voice recognition. Some of the recent applications include:

- Predicting COVID-19 resource needs from a series of X-rays (`https://arxiv.org/abs/2101.04909`)
- Modeling virus mutations (`https://science.sciencemag.org/content/371/6526/284`)
- Leveraging data from social media platforms to manage extreme weather events (`https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-5973.12311`)
- Improving photo descriptions for people who are blind or visually impaired (`https://tech.fb.com/how-facebook-is-using-ai-to-improve-photo-descriptions-for-people-who-are-blind-or-visually-impaired/`)

Single-layer neural network recap

This chapter is all about multilayer NNs, how they work, and how to train them to solve complex problems. However, before we dig deeper into a particular multilayer NN architecture, let's briefly reiterate some of the concepts of single-layer NNs that we introduced in *Chapter 2*, namely, the **ADAptive LInear NEuron** (**Adaline**) algorithm, which is shown in *Figure 11.1*:

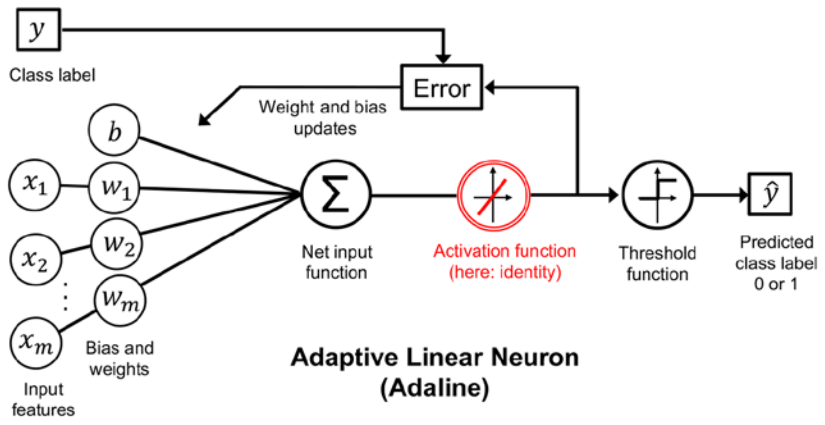

Figure 11.1: The Adaline algorithm

In *Chapter 2*, we implemented the Adaline algorithm to perform binary classification, and we used the gradient descent optimization algorithm to learn the weight coefficients of the model. In every epoch (pass over the training dataset), we updated the weight vector $\boldsymbol{w}$ and bias unit b using the following update rule:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \quad b := b + \Delta b$$

where $\Delta w_j = -\eta \frac{\partial L}{\partial w_j}$ and $\Delta b = -\eta \frac{\partial L}{\partial b}$ for the bias unit and each weight w_j in the weight vector $\boldsymbol{w}$.

In other words, we computed the gradient based on the whole training dataset and updated the weights of the model by taking a step in the opposite direction of the loss gradient $\nabla L(\boldsymbol{w})$. (For simplicity, we will focus on the weights and omit the bias unit in the following paragraphs; however, as you remember from *Chapter 2*, the same concepts apply.) In order to find the optimal weights of the model, we optimized an objective function that we defined as the **mean of squared errors** (**MSE**) loss function $L(\boldsymbol{w})$. Furthermore, we multiplied the gradient by a factor, the **learning rate** η, which we had to choose carefully to balance the speed of learning against the risk of overshooting the global minimum of the loss function.

In gradient descent optimization, we updated all weights simultaneously after each epoch, and we defined the partial derivative for each weight w_j in the weight vector, $\boldsymbol{w}$, as follows:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{n}\sum_i \left(y^{(i)} - a^{(i)}\right)^2 = -\frac{2}{n}\sum_i \left(y^{(i)} - a^{(i)}\right)x_j^{(i)}$$

Here, $y^{(i)}$ is the target class label of a particular sample $x^{(i)}$, and $a^{(i)}$ is the activation of the neuron, which is a linear function in the special case of Adaline.

Furthermore, we defined the activation function $\sigma(\cdot)$ as follows:

$$\sigma(\cdot) = z = a$$

Here, the net input, z, is a linear combination of the weights that are connecting the input layer to the output layer:

$$z = \sum_j w_j x_j + b = \boldsymbol{w}^T \boldsymbol{x} + b$$

While we used the activation $\sigma(\cdot)$ to compute the gradient update, we implemented a threshold function to squash the continuous-valued output into binary class labels for prediction:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0; \\ 0 & \text{otherwise} \end{cases}$$

Single-layer naming convention

Note that although Adaline consists of two layers, one input layer and one output layer, it is called a single-layer network because of its single link between the input and output layers.

Also, we learned about a certain *trick* to accelerate the model learning, the so-called **stochastic gradient descent** (**SGD**) optimization. SGD approximates the loss from a single training sample (online learning) or a small subset of training examples (mini-batch learning). We will make use of this concept later in this chapter when we implement and train a **multilayer perceptron** (**MLP**). Apart from faster learning—due to the more frequent weight updates compared to gradient descent—its noisy nature is also regarded as beneficial when training multilayer NNs with nonlinear activation functions, which do not have a convex loss function. Here, the added noise can help to escape local loss minima, but we will discuss this topic in more detail later in this chapter.

Introducing the multilayer neural network architecture

In this section, you will learn how to connect multiple single neurons to a multilayer feedforward NN; this special type of *fully connected* network is also called **MLP**.

Figure 11.2 illustrates the concept of an MLP consisting of two layers:

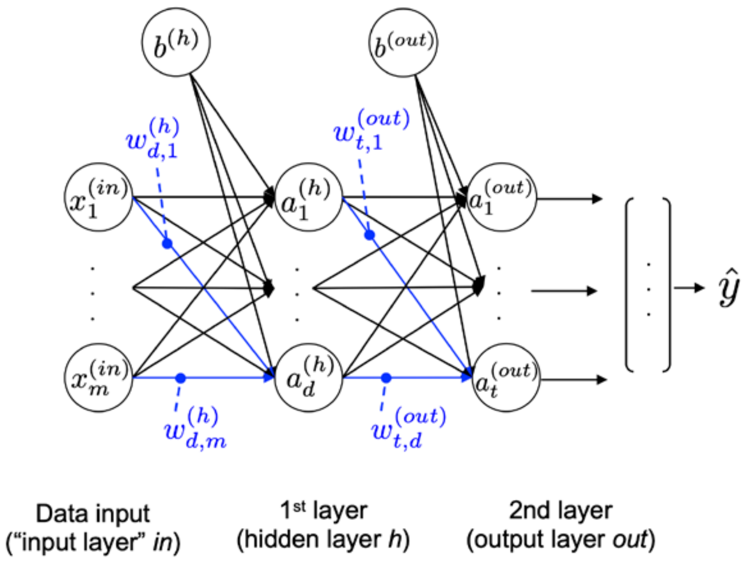

Figure 11.2: A two-layer MLP

Next to the data input, the MLP depicted in *Figure 11.2* has one hidden layer and one output layer. The units in the hidden layer are fully connected to the input features, and the output layer is fully connected to the hidden layer. If such a network has more than one hidden layer, we also call it a **deep NN**. (Note that in some contexts, the inputs are also regarded as a layer. However, in this case, it would make the Adaline model, which is a single-layer neural network, a two-layer neural network, which may be counterintuitive.)

Adding additional hidden layers

We can add any number of hidden layers to the MLP to create deeper network architectures. Practically, we can think of the number of layers and units in an NN as additional hyperparameters that we want to optimize for a given problem task using the cross-validation technique, which we discussed in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

However, the loss gradients for updating the network's parameters, which we will calculate later via backpropagation, will become increasingly small as more layers are added to a network. This vanishing gradient problem makes model learning more challenging. Therefore, special algorithms have been developed to help train such DNN structures; this is known as **deep learning**, which we will discuss in more detail in the following chapters.

As shown in *Figure 11.2*, we denote the ith activation unit in the lth layer as $a_i^{(l)}$. To make the math and code implementations a bit more intuitive, we will not use numerical indices to refer to layers, but we will use the *in* superscript for the input features, the h superscript for the hidden layer, and the *out* superscript for the output layer. For instance, $x_i^{(in)}$ refers to the ith input feature value, $a_i^{(h)}$ refers to the ith unit in the hidden layer, and $a_i^{(out)}$ refers to the ith unit in the output layer. Note that the $\boldsymbol{b}$'s in *Figure 11.2* denote the bias units. In fact, $\boldsymbol{b}^{(h)}$ and $\boldsymbol{b}^{(out)}$ are vectors with the number of elements being equal to the number of nodes in the layer they correspond to. For example, $\boldsymbol{b}^{(h)}$ stores d bias units, where d is the number of nodes in the hidden layer. If this sounds confusing, don't worry. Looking at the code implementation later, where we initialize weight matrices and bias unit vectors, will help clarify these concepts.

Each node in layer l is connected to all nodes in layer $l + 1$ via a weight coefficient. For example, the connection between the kth unit in layer l to the jth unit in layer $l + 1$ will be written as $w_{j,k}^{(l)}$. Referring back to *Figure 11.2*, we denote the weight matrix that connects the input to the hidden layer as $\boldsymbol{W}^{(h)}$, and we write the matrix that connects the hidden layer to the output layer as $\boldsymbol{W}^{(out)}$.

While one unit in the output layer would suffice for a binary classification task, we saw a more general form of an NN in the preceding figure, which allows us to perform multiclass classification via a generalization of the **one-versus-all** (**OvA**) technique. To better understand how this works, remember the **one-hot** representation of categorical variables that we introduced in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*.

For example, we can encode the three class labels in the familiar Iris dataset (0=*Setosa*, 1=*Versicolor*, 2=*Virginica*) as follows:

$$0 = \begin{bmatrix}1\\0\\0\end{bmatrix}, 1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, 2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

This one-hot vector representation allows us to tackle classification tasks with an arbitrary number of unique class labels present in the training dataset.

If you are new to NN representations, the indexing notation (subscripts and superscripts) may look a little bit confusing at first. What may seem overly complicated at first will make much more sense in later sections when we vectorize the NN representation. As introduced earlier, we summarize the weights that connect the input and hidden layers by a $d \times m$ dimensional matrix $\boldsymbol{W}^{(h)}$, where d is the number of hidden units and m is the number of input units.

Activating a neural network via forward propagation

In this section, we will describe the process of **forward propagation** to calculate the output of an MLP model. To understand how it fits into the context of learning an MLP model, let's summarize the MLP learning procedure in three simple steps:

1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output.
2. Based on the network's output, we calculate the loss that we want to minimize using a loss function that we will describe later.

3. We backpropagate the loss, find its derivative with respect to each weight and bias unit in the network, and update the model.

Finally, after we repeat these three steps for multiple epochs and learn the weight and bias parameters of the MLP, we use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in the one-hot representation, which we described in the previous section.

Now, let's walk through the individual steps of forward propagation to generate an output from the patterns in the training data. Since each unit in the hidden layer is connected to all units in the input layers, we first calculate the activation unit of the hidden layer $a_1^{(h)}$ as follows:

$$z_1^{(h)} = x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \cdots + x_m^{(in)} w_{1,m}^{(h)}$$

$$a_1^{(h)} = \sigma\left(z_1^{(h)}\right)$$

Here, $z_1^{(h)}$ is the net input and $\sigma(\cdot)$ is the activation function, which has to be differentiable to learn the weights that connect the neurons using a gradient-based approach. To be able to solve complex problems such as image classification, we need nonlinear activation functions in our MLP model, for example, the sigmoid (logistic) activation function that we remember from the section about logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

As you may recall, the sigmoid function is an *S*-shaped curve that maps the net input *z* onto a logistic distribution in the range 0 to 1, which cuts the *y* axis at *z* = 0, as shown in *Figure 11.3*:

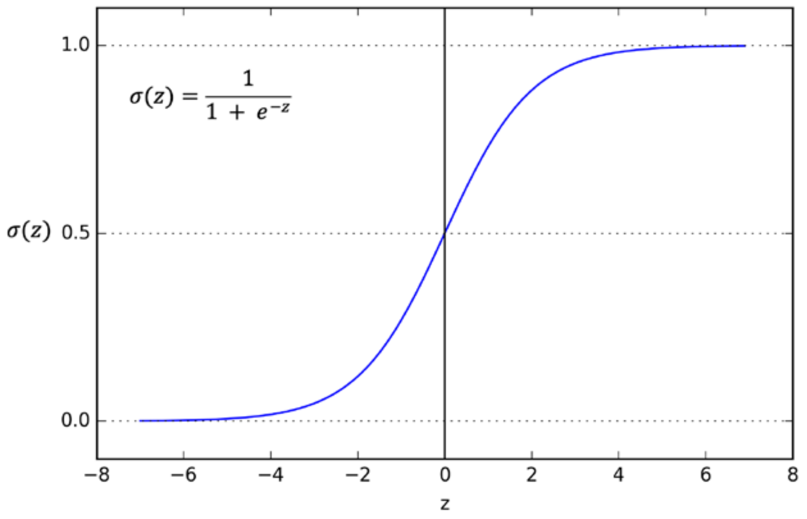

Figure 11.3: The sigmoid activation function

MLP is a typical example of a feedforward artificial NN. The term **feedforward** refers to the fact that each layer serves as the input to the next layer without loops, in contrast to recurrent NNs—an architecture that we will discuss later in this chapter and discuss in more detail in *Chapter 15, Modeling Sequential Data Using Recurrent Neural Networks*. The term *multilayer perceptron* may sound a little bit confusing since the artificial neurons in this network architecture are typically sigmoid units, not perceptrons. We can think of the neurons in the MLP as logistic regression units that return values in the continuous range between 0 and 1.

For purposes of code efficiency and readability, we will now write the activation in a more compact form using the concepts of basic linear algebra, which will allow us to vectorize our code implementation via NumPy rather than writing multiple nested and computationally expensive Python `for` loops:

$$z^{(h)} = \boldsymbol{x}^{(in)}\boldsymbol{W}^{(h)T} + \boldsymbol{b}^{(h)}$$

$$a^{(h)} = \sigma\left(\boldsymbol{z}^{(h)}\right)$$

Here, $\boldsymbol{z}^{(h)}$ is our 1×m dimensional feature vector. $\boldsymbol{W}^{(h)}$ is a d×m dimensional weight matrix where d is the number of units in the hidden layer; consequently, the transposed matrix $\boldsymbol{W}^{(h)T}$ is m×d dimensional. The bias vector $\boldsymbol{b}^{(h)}$ consists of d bias units (one bias unit per hidden node).

After matrix-vector multiplication, we obtain the 1×d dimensional net input vector $\boldsymbol{z}^{(h)}$ to calculate the activation $\boldsymbol{a}^{(h)}$ (where $\boldsymbol{a}^{(h)} \in \mathbb{R}^{1\times d}$).

Furthermore, we can generalize this computation to all n examples in the training dataset:

$$\boldsymbol{Z}^{(h)} = \boldsymbol{X}^{(in)}\boldsymbol{W}^{(h)T} + \boldsymbol{b}^{(h)}$$

Here, $\boldsymbol{X}^{(in)}$ is now an n×m matrix, and the matrix multiplication will result in an n×d dimensional net input matrix, $\boldsymbol{Z}^{(h)}$. Finally, we apply the activation function $\sigma(\cdot)$ to each value in the net input matrix to get the n×d activation matrix in the next layer (here, the output layer):

$$\boldsymbol{A}^{(h)} = \sigma\left(\boldsymbol{Z}^{(h)}\right)$$

Similarly, we can write the activation of the output layer in vectorized form for multiple examples:

$$\boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)T} + \boldsymbol{b}^{(out)}$$

Here, we multiply the transpose of the t×d matrix $\boldsymbol{W}^{(out)}$ (t is the number of output units) by the n×d dimensional matrix, $\boldsymbol{A}^{(h)}$, and add the t dimensional bias vector $\boldsymbol{b}^{(out)}$ to obtain the n×t dimensional matrix, $\boldsymbol{Z}^{(out)}$. (The columns in this matrix represent the outputs for each sample.)

Lastly, we apply the sigmoid activation function to obtain the continuous-valued output of our network:

$$\boldsymbol{A}^{(out)} = \sigma\left(\boldsymbol{Z}^{(out)}\right)$$

Similar to $\boldsymbol{Z}^{(out)}$, $\boldsymbol{A}^{(out)}$ is an n×t dimensional matrix.

Classifying handwritten digits

In the previous section, we covered a lot of the theory around NNs, which can be a little bit overwhelming if you are new to this topic. Before we continue with the discussion of the algorithm for learning the weights of the MLP model, backpropagation, let's take a short break from the theory and see an NN in action.

Additional resources on backpropagation

The NN theory can be quite complex; thus, we want to provide readers with additional resources that cover some of the topics we discuss in this chapter in more detail or from a different perspective:

- *Chapter 6*, *Deep Feedforward Networks*, *Deep Learning*, by *I. Goodfellow*, *Y. Bengio*, and *A. Courville*, MIT Press, 2016 (manuscripts freely accessible at `http://www.deeplearningbook.org`).
- *Pattern Recognition and Machine Learning*, by *C. M. Bishop*, Springer New York, 2006.
- Lecture video slides from Sebastian Raschka's deep learning course:

  `https://sebastianraschka.com/blog/2021/dl-course.html#l08-multinomial-logistic-regression--softmax-regression`

  `https://sebastianraschka.com/blog/2021/dl-course.html#l09-multilayer-perceptrons-and-backpropration`

In this section, we will implement and train our first multilayer NN to classify handwritten digits from the popular **Mixed National Institute of Standards and Technology** (**MNIST**) dataset that has been constructed by Yann LeCun and others and serves as a popular benchmark dataset for machine learning algorithms (*Gradient-Based Learning Applied to Document Recognition* by *Y. LeCun*, *L. Bottou*, *Y. Bengio*, and *P. Haffner*, *Proceedings of the IEEE*, 86(11): 2278-2324, 1998).

Obtaining and preparing the MNIST dataset

The MNIST dataset is publicly available at `http://yann.lecun.com/exdb/mnist/` and consists of the following four parts:

1. **Training dataset images:** `train-images-idx3-ubyte.gz` (9.9 MB, 47 MB unzipped, and 60,000 examples)
2. **Training dataset labels:** `train-labels-idx1-ubyte.gz` (29 KB, 60 KB unzipped, and 60,000 labels)
3. **Test dataset images:** `t10k-images-idx3-ubyte.gz` (1.6 MB, 7.8 MB unzipped, and 10,000 examples)
4. **Test dataset labels:** `t10k-labels-idx1-ubyte.gz` (5 KB, 10 KB unzipped, and 10,000 labels)

The MNIST dataset was constructed from two datasets of the US **National Institute of Standards and Technology** (**NIST**). The training dataset consists of handwritten digits from 250 different people, 50 percent high school students, and 50 percent employees from the Census Bureau. Note that the test dataset contains handwritten digits from different people following the same split.

Instead of downloading the abovementioned dataset files and preprocessing them into NumPy arrays ourselves, we will use scikit-learn's new `fetch_openml` function, which allows us to load the MNIST dataset more conveniently:

```
>>> from sklearn.datasets import fetch_openml
>>> X, y = fetch_openml('mnist_784', version=1,
...                     return_X_y=True)
>>> X = X.values
>>> y = y.astype(int).values
```

In scikit-learn, the `fetch_openml` function downloads the MNIST dataset from OpenML (`https://www.openml.org/d/554`) as pandas `DataFrame` and Series objects, which is why we use the `.values` attribute to obtain the underlying NumPy arrays. (If you are using a scikit-learn version older than 1.0, `fetch_openml` downloads NumPy arrays directly so you can omit using the `.values` attribute.) The $n \times m$ dimensional `X` array consists of 70,000 images with 784 pixels each, and the `y` array stores the corresponding 70,000 class labels, which we can confirm by checking the dimensions of the arrays as follows:

```
>>> print(X.shape)
(70000, 784)
>>> print(y.shape)
(70000,)
```

The images in the MNIST dataset consist of 28×28 pixels, and each pixel is represented by a grayscale intensity value. Here, `fetch_openml` already unrolled the 28×28 pixels into one-dimensional row vectors, which represent the rows in our `X` array (784 per row or image) above. The second array (`y`) returned by the `fetch_openml` function contains the corresponding target variable, the class labels (integers 0-9) of the handwritten digits.

Next, let's normalize the pixels values in MNIST to the range –1 to 1 (originally 0 to 255) via the following code line:

```
>>> X = ((X / 255.) - .5) * 2
```

The reason behind this is that gradient-based optimization is much more stable under these conditions, as discussed in *Chapter 2*. Note that we scaled the images on a pixel-by-pixel basis, which is different from the feature-scaling approach that we took in previous chapters.

Previously, we derived scaling parameters from the training dataset and used these to scale each column in the training dataset and test dataset. However, when working with image pixels, centering them at zero and rescaling them to a [–1, 1] range is also common and usually works well in practice.

To get an idea of how those images in MNIST look, let's visualize examples of the digits 0-9 after reshaping the 784-pixel vectors from our feature matrix into the original 28×28 image that we can plot via Matplotlib's `imshow` function:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5,
...                        sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X[y == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

We should now see a plot of the 2×5 subfigures showing a representative image of each unique digit:

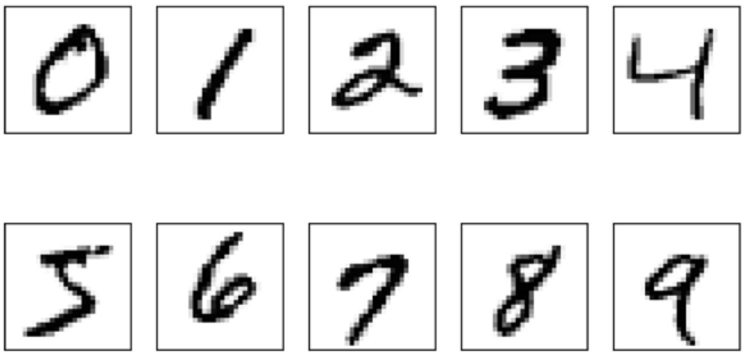

Figure 11.4: A plot showing one randomly chosen handwritten digit from each class

In addition, let's also plot multiple examples of the same digit to see how different the handwriting for each really is:

```
>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X[y == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we should now see the first 25 variants of the digit 7:

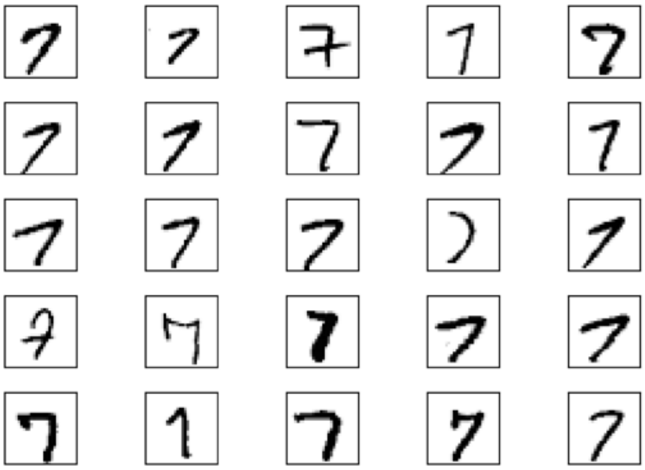

Figure 11.5: Different variants of the handwritten digit 7

Finally, let's divide the dataset into training, validation, and test subsets. The following code will split the dataset such that 55,000 images are used for training, 5,000 images for validation, and 10,000 images for testing:

```
>>> from sklearn.model_selection import train_test_split
>>> X_temp, X_test, y_temp, y_test = train_test_split(
...     X, y, test_size=10000, random_state=123, stratify=y
... )
>>> X_train, X_valid, y_train, y_valid = train_test_split(
...     X_temp, y_temp, test_size=5000,
...     random_state=123, stratify=y_temp
... )
```

Implementing a multilayer perceptron

In this subsection, we will now implement an MLP from scratch to classify the images in the MNIST dataset. To keep things simple, we will implement an MLP with only one hidden layer. Since the approach may seem a little bit complicated at first, you are encouraged to download the sample code for this chapter from the Packt Publishing website or from GitHub (`https://github.com/rasbt/machine-learning-book`) so that you can view this MLP implementation annotated with comments and syntax highlighting for better readability.

If you are not running the code from the accompanying Jupyter Notebook file or don't have access to the internet, copy the `NeuralNetMLP` code from this chapter into a Python script file in your current working directory (for example, `neuralnet.py`), which you can then import into your current Python session via the following command:

```
from neuralnet import NeuralNetMLP
```

The code will contain parts that we have not talked about yet, such as the backpropagation algorithm. Do not worry if not all the code makes immediate sense to you; we will follow up on certain parts later in this chapter. However, going over the code at this stage can make it easier to follow the theory later.

So, let's look at the following implementation of an MLP, starting with the two helper functions to compute the logistic sigmoid activation and to convert integer class label arrays to one-hot encoded labels:

```
import numpy as np

def sigmoid(z):
    return 1. / (1. + np.exp(-z))


def int_to_onehot(y, num_labels):

    ary = np.zeros((y.shape[0], num_labels))
    for i, val in enumerate(y):
        ary[i, val] = 1

    return ary
```

Below, we implement the main class for our MLP, which we call `NeuralNetMLP`. There are three class methods, `.__init__()`, `.forward()`, and `.backward()`, that we will discuss one by one, starting with the `__init__` constructor:

```
class NeuralNetMLP:

    def __init__(self, num_features, num_hidden,
                 num_classes, random_seed=123):
        super().__init__()

        self.num_classes = num_classes

        # hidden
        rng = np.random.RandomState(random_seed)

        self.weight_h = rng.normal(
            loc=0.0, scale=0.1, size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        # output
        self.weight_out = rng.normal(
            loc=0.0, scale=0.1, size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)
```

The `__init__` constructor instantiates the weight matrices and bias vectors for the hidden and the output layer. Next, let's see how these are used in the `forward` method to make predictions:

```
    def forward(self, x):
        # Hidden layer

        # input dim: [n_hidden, n_features]
        #        dot [n_features, n_examples] .T
        # output dim: [n_examples, n_hidden]
        z_h = np.dot(x, self.weight_h.T) + self.bias_h
        a_h = sigmoid(z_h)

        # Output layer
        # input dim: [n_classes, n_hidden]
        #        dot [n_hidden, n_examples] .T
        # output dim: [n_examples, n_classes]
        z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
        a_out = sigmoid(z_out)
        return a_h, a_out
```

The forward method takes in one or more training examples and returns the predictions. In fact, it returns both the activation values from the hidden layer and the output layer, a_h and a_out. While a_out represents the class-membership probabilities that we can convert to class labels, which we care about, we also need the activation values from the hidden layer, a_h, to optimize the model parameters; that is, the weight and bias units of the hidden and output layers.

Finally, let's talk about the backward method, which updates the weight and bias parameters of the neural network:

```
    def backward(self, x, a_h, a_out, y):

        #########################
        ### Output layer weights
        #########################

        # one-hot encoding
        y_onehot = int_to_onehot(y, self.num_classes)

        # Part 1: dLoss/dOutWeights
        ## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
        ## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet
        ## for convenient re-use

        # input/output dim: [n_examples, n_classes]
        d_loss__d_a_out = 2.*(a_out - y_onehot) / y.shape[0]

        # input/output dim: [n_examples, n_classes]
        d_a_out__d_z_out = a_out * (1. - a_out) # sigmoid derivative

        # output dim: [n_examples, n_classes]
        delta_out = d_loss__d_a_out * d_a_out__d_z_out

        # gradient for output weights

        # [n_examples, n_hidden]
        d_z_out__dw_out = a_h

        # input dim: [n_classes, n_examples]
        #            dot [n_examples, n_hidden]
        # output dim: [n_classes, n_hidden]
        d_loss__dw_out = np.dot(delta_out.T, d_z_out__dw_out)
        d_loss__db_out = np.sum(delta_out, axis=0)
        #################################
        # Part 2: dLoss/dHiddenWeights
        ## = DeltaOut * dOutNet/dHiddenAct * dHiddenAct/dHiddenNet
        #    * dHiddenNet/dWeight

        # [n_classes, n_hidden]
        d_z_out__a_h = self.weight_out

        # output dim: [n_examples, n_hidden]
        d_loss__a_h = np.dot(delta_out, d_z_out__a_h)

        # [n_examples, n_hidden]
        d_a_h__d_z_h = a_h * (1. - a_h) # sigmoid derivative

        # [n_examples, n_features]
        d_z_h__d_w_h = x

        # output dim: [n_hidden, n_features]
        d_loss__d_w_h = np.dot((d_loss__a_h * d_a_h__d_z_h).T,
                               d_z_h__d_w_h)
        d_loss__d_b_h = np.sum((d_loss__a_h * d_a_h__d_z_h), axis=0)

        return (d_loss__dw_out, d_loss__db_out,
                d_loss__d_w_h, d_loss__d_b_h)
```

The `backward` method implements the so-called *backpropagation* algorithm, which calculates the gradients of the loss with respect to the weight and bias parameters. Similar to Adaline, these gradients are then used to update these parameters via gradient descent. Note that multilayer NNs are more complex than their single-layer siblings, and we will go over the mathematical concepts of how to compute the gradients in a later section after discussing the code. For now, just consider the `backward` method as a way for computing gradients that are used for the gradient descent updates. For simplicity, the loss function this derivation is based on is the same MSE loss that we used in Adaline. In later chapters, we will look at alternative loss functions, such as multi-category cross-entropy loss, which is a generalization of the binary logistic regression loss to multiple classes.

Looking at this code implementation of the `NeuralNetMLP` class, you may have noticed that this object-oriented implementation differs from the familiar scikit-learn API that is centered around the `.fit()` and `.predict()` methods. Instead, the main methods of the `NeuralNetMLP` class are the `.forward()` and `.backward()` methods. One of the reasons behind this is that it makes a complex neural network a bit easier to understand in terms of how the information flows through the networks.

Another reason is that this implementation is relatively similar to how more advanced deep learning libraries such as PyTorch operate, which we will introduce and use in the upcoming chapters to implement more complex neural networks.

After we have implemented the `NeuralNetMLP` class, we use the following code to instantiate a new `NeuralNetMLP` object:

```
>>> model = NeuralNetMLP(num_features=28*28,
...                      num_hidden=50,
...                      num_classes=10)
```

The `model` accepts MNIST images reshaped into 784-dimensional vectors (in the format of `X_train`, `X_valid`, or `X_test`, which we defined previously) for the 10 integer classes (digits 0-9). The hidden layer consists of 50 nodes. Also, as you may be able to tell from looking at the previously defined `.forward()` method, we use a sigmoid activation function after the first hidden layer and output layer to keep things simple. In later chapters, we will learn about alternative activation functions for both the hidden and output layers.

Figure 11.6 summarizes the neural network architecture that we instantiated above:

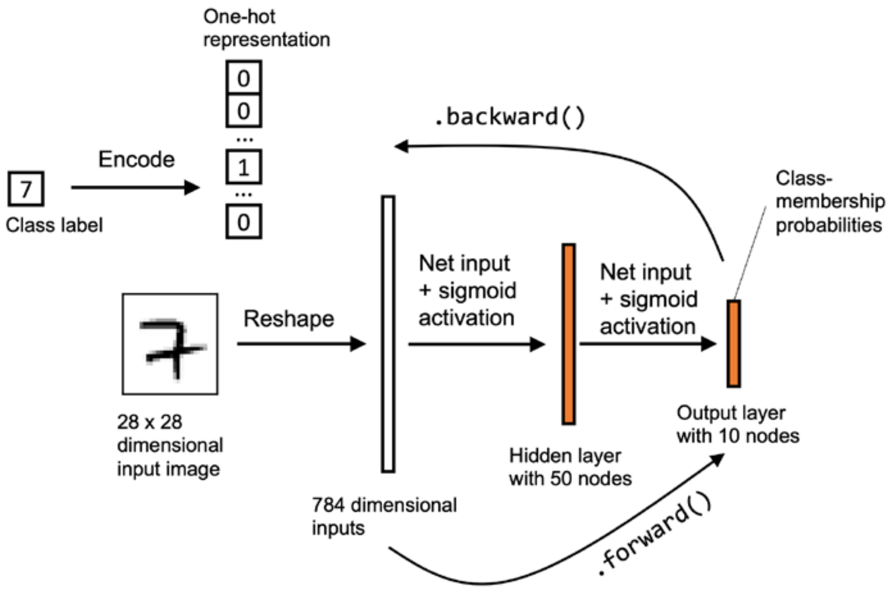
One-hot representation
0
0
1
0
Encode
7
Class label
.backward()
Class-membership probabilities
Reshape
Net input + sigmoid activation
Net input + sigmoid activation
28 x 28 dimensional input image
784 dimensional inputs
Hidden layer with 50 nodes
Output layer with 10 nodes
.forward()

Figure 11.6: The NN architecture for labeling handwritten digits

In the next subsection, we are going to implement the training function that we can use to train the network on mini-batches of the data via backpropagation.

Coding the neural network training loop

Now that we have implemented the `NeuralNetMLP` class in the previous subsection and initiated a model, the next step is to train the model. We will tackle this in multiple steps. First, we will define some helper functions for data loading. Second, we will embed these functions into the training loop that iterates over the dataset in multiple epochs.

The first function we are going to define is a mini-batch generator, which takes in our dataset and divides it into mini-batches of a desired size for stochastic gradient descent training. The code is as follows:

```
>>> import numpy as np
>>> num_epochs = 50
>>> minibatch_size = 100

>>> def minibatch_generator(X, y, minibatch_size):
...     indices = np.arange(X.shape[0])
...     np.random.shuffle(indices)
...     for start_idx in range(0, indices.shape[0] - minibatch_size
...                            + 1, minibatch_size):
...         batch_idx = indices[start_idx:start_idx + minibatch_size]
...         yield X[batch_idx], y[batch_idx]
```

Before we move on to the next functions, let's confirm that the mini-batch generator works as intended and produces mini-batches of the desired size. The following code will attempt to iterate through the dataset, and then we will print the dimension of the mini-batches. Note that in the following code examples, we will remove the `break` statements. The code is as follows:

```
>>> # iterate over training epochs
>>> for i in range(num_epochs):
...     # iterate over minibatches
...     minibatch_gen = minibatch_generator(
...         X_train, y_train, minibatch_size)
...     for X_train_mini, y_train_mini in minibatch_gen:
...         break
...     break
>>> print(X_train_mini.shape)
(100, 784)
>>> print(y_train_mini.shape)
(100,)
```

As we can see, the network returns mini-batches of size 100 as intended.

Next, we have to define our loss function and performance metric that we can use to monitor the training process and evaluate the model. The MSE loss and accuracy function can be implemented as follows:

```
>>> def mse_loss(targets, probas, num_labels=10):
...     onehot_targets = int_to_onehot(
...         targets, num_labels=num_labels
...     )
...     return np.mean((onehot_targets - probas)**2)

>>> def accuracy(targets, predicted_labels):
...     return np.mean(predicted_labels == targets)
```

Let's test the preceding function and compute the initial validation set MSE and accuracy of the model we instantiated in the previous section:

```
>>> _, probas = model.forward(X_valid)
>>> mse = mse_loss(y_valid, probas)
>>> print(f'Initial validation MSE: {mse:.1f}')
Initial validation MSE: 0.3

>>> predicted_labels = np.argmax(probas, axis=1)
>>> acc = accuracy(y_valid, predicted_labels)
>>> print(f'Initial validation accuracy: {acc*100:.1f}%')
Initial validation accuracy: 9.4%
```

In this code example, note that `model.forward()` returns the hidden and output layer activations. Remember that we have 10 output nodes (one corresponding to each unique class label). Hence, when computing the MSE, we first converted the class labels into one-hot encoded class labels in the `mse_loss()` function. In practice, it does not make a difference whether we average over the row or the columns of the squared-difference matrix first, so we simply call `np.mean()` without any axis specification so that it returns a scalar.

The output layer activations, since we used the logistic sigmoid function, are values in the range [0, 1]. For each input, the output layer produces 10 values in the range [0, 1], so we used the `np.argmax()` function to select the index position of the largest value, which yields the predicted class label. We then compared the true labels with the predicted class labels to compute the accuracy via the `accuracy()` function we defined. As we can see from the preceding output, the accuracy is not very high. However, given that we have a balanced dataset with 10 classes, a prediction accuracy of approximately 10 percent is what we would expect for an untrained model producing random predictions.

Using the previous code, we can compute the performance on, for example, the whole training set if we provide `y_train` as input to targets and the predicted labels from feeding the model with `X_train`. However, in practice, our computer memory is usually a limiting factor for how much data the model can ingest in one forward pass (due to the large matrix multiplications). Hence, we are defining our MSE and accuracy computation based on our previous mini-batch generator. The following function will compute the MSE and accuracy incrementally by iterating over the dataset one mini-batch at a time to be more memory-efficient:

```
>>> def compute_mse_and_acc(nnet, X, y, num_labels=10,
...                         minibatch_size=100):
...     mse, correct_pred, num_examples = 0., 0, 0
...     minibatch_gen = minibatch_generator(X, y, minibatch_size)
...     for i, (features, targets) in enumerate(minibatch_gen):
...         _, probas = nnet.forward(features)
...         predicted_labels = np.argmax(probas, axis=1)
...         onehot_targets = int_to_onehot(
...             targets, num_labels=num_labels
...         )
...         loss = np.mean((onehot_targets - probas)**2)
...         correct_pred += (predicted_labels == targets).sum()
...         num_examples += targets.shape[0]
...         mse += loss
...     mse = mse/i
...     acc = correct_pred/num_examples
...     return mse, acc
```

Before we implement the training loop, let's test the function and compute the initial training set MSE and accuracy of the model we instantiated in the previous section and make sure it works as intended:

```
>>> mse, acc = compute_mse_and_acc(model, X_valid, y_valid)
>>> print(f'Initial valid MSE: {mse:.1f}')
Initial valid MSE: 0.3
>>> print(f'Initial valid accuracy: {acc*100:.1f}%')
Initial valid accuracy: 9.4%
```

As we can see from the results, our generator approach produces the same results as the previously defined MSE and accuracy functions, except for a small rounding error in the MSE (0.27 versus 0.28), which is negligible for our purposes.

Let's now get to the main part and implement the code to train our model:

```
>>> def train(model, X_train, y_train, X_valid, y_valid, num_epochs,
...           learning_rate=0.1):
...     epoch_loss = []
...     epoch_train_acc = []
...     epoch_valid_acc = []
...
...     for e in range(num_epochs):
...         # iterate over minibatches
...         minibatch_gen = minibatch_generator(
...             X_train, y_train, minibatch_size)
...         for X_train_mini, y_train_mini in minibatch_gen:
...             #### Compute outputs ####
...             a_h, a_out = model.forward(X_train_mini)

...             #### Compute gradients ####
...             d_loss__d_w_out, d_loss__d_b_out, \
...             d_loss__d_w_h, d_loss__d_b_h = \
...                 model.backward(X_train_mini, a_h, a_out,
...                                y_train_mini)
...
...             #### Update weights ####
...             model.weight_h -= learning_rate * d_loss__d_w_h
...             model.bias_h -= learning_rate * d_loss__d_b_h
...             model.weight_out -= learning_rate * d_loss__d_w_out
...             model.bias_out -= learning_rate * d_loss__d_b_out
...
...         #### Epoch Logging ####
...         train_mse, train_acc = compute_mse_and_acc(
...             model, X_train, y_train
...         )
...         valid_mse, valid_acc = compute_mse_and_acc(
...             model, X_valid, y_valid
...         )
...         train_acc, valid_acc = train_acc*100, valid_acc*100
...         epoch_train_acc.append(train_acc)
...         epoch_valid_acc.append(valid_acc)
...         epoch_loss.append(train_mse)
...         print(f'Epoch: {e+1:03d}/{num_epochs:03d} '
...               f'| Train MSE: {train_mse:.2f} '
...               f'| Train Acc: {train_acc:.2f}% '
...               f'| Valid Acc: {valid_acc:.2f}%')
...
...     return epoch_loss, epoch_train_acc, epoch_valid_acc
```

On a high level, the `train()` function iterates over multiple epochs, and in each epoch, it used the previously defined `minibatch_generator()` function to iterate over the whole training set in mini-batches for stochastic gradient descent training. Inside the mini-batch generator `for` loop, we obtain the outputs from the model, `a_h` and `a_out`, via its `.forward()` method. Then, we compute the loss gradients via the model's `.backward()` method—the theory will be explained in a later section. Using the loss gradients, we update the weights by adding the negative gradient multiplied by the learning rate. This is the same concept that we discussed earlier for Adaline. For example, to update the model weights of the hidden layer, we defined the following line:

```
model.weight_h -= learning_rate * d_loss__d_w_h
```

For a single weight, w_j, this corresponds to the following partial derivative-based update:

$$w_j \coloneqq w_j - \eta \frac{\partial L}{\partial w_j}$$

Finally, the last portion of the previous code computes the losses and prediction accuracies on the training and test sets to track the training progress.

Let's now execute this function to train our model for 50 epochs, which may take a few minutes to finish:

```
>>> np.random.seed(123) # for the training set shuffling
>>> epoch_loss, epoch_train_acc, epoch_valid_acc = train(
...     model, X_train, y_train, X_valid, y_valid,
...     num_epochs=50, learning_rate=0.1)
```

During training, we should see the following output:

```
Epoch: 001/050 | Train MSE: 0.05 | Train Acc: 76.17% | Valid Acc: 76.02%
Epoch: 002/050 | Train MSE: 0.03 | Train Acc: 85.46% | Valid Acc: 84.94%
Epoch: 003/050 | Train MSE: 0.02 | Train Acc: 87.89% | Valid Acc: 87.64%
Epoch: 004/050 | Train MSE: 0.02 | Train Acc: 89.36% | Valid Acc: 89.38%
Epoch: 005/050 | Train MSE: 0.02 | Train Acc: 90.21% | Valid Acc: 90.16%
...
Epoch: 048/050 | Train MSE: 0.01 | Train Acc: 95.57% | Valid Acc: 94.58%
Epoch: 049/050 | Train MSE: 0.01 | Train Acc: 95.55% | Valid Acc: 94.54%
Epoch: 050/050 | Train MSE: 0.01 | Train Acc: 95.59% | Valid Acc: 94.74%
```

The reason why we print all this output is that, in NN training, it is really useful to compare training and validation accuracy. This helps us judge whether the network model performs well, given the architecture and hyperparameters. For example, if we observe a low training and validation accuracy, there is likely an issue with the training dataset, or the hyperparameters' settings are not ideal.

In general, training (deep) NNs is relatively expensive compared with the other models we've discussed so far. Thus, we want to stop it early in certain circumstances and start over with different hyperparameter settings. On the other hand, if we find that it increasingly tends to overfit the training data (noticeable by an increasing gap between training and validation dataset performance), we may want to stop the training early, as well.

In the next subsection, we will discuss the performance of our NN model in more detail.

Evaluating the neural network performance

Before we discuss backpropagation, the training procedure of NNs, in more detail in the next section, let's look at the performance of the model that we trained in the previous subsection.

In `train()`, we collected the training loss and the training and validation accuracy for each epoch so that we can visualize the results using Matplotlib. Let's look at the training MSE loss first:

```
>>> plt.plot(range(len(epoch_loss)), epoch_loss)
>>> plt.ylabel('Mean squared error')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

The preceding code plots the loss over the 50 epochs, as shown in *Figure 11.7*:

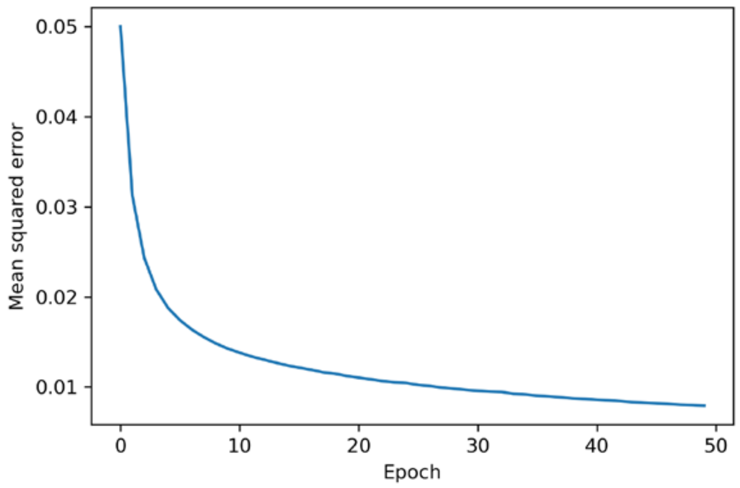

Figure 11.7: A plot of the MSE by the number of training epochs

As we can see, the loss decreased substantially during the first 10 epochs and seems to slowly converge in the last 10 epochs. However, the small slope between epoch 40 and epoch 50 indicates that the loss would further decrease with training over additional epochs.

Next, let's take a look at the training and validation accuracy:

```
>>> plt.plot(range(len(epoch_train_acc)), epoch_train_acc,
...          label='Training')
>>> plt.plot(range(len(epoch_valid_acc)), epoch_valid_acc,
...          label='Validation')
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Epochs')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

The preceding code examples plot those accuracy values over the 50 training epochs, as shown in *Figure 11.8*:

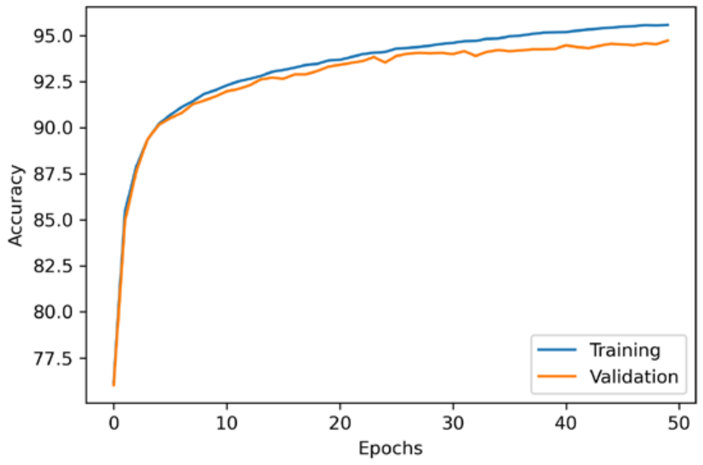

Figure 11.8: Classification accuracy by the number of training epochs

The plot reveals that the gap between training and validation accuracy increases as we train for more epochs. At approximately the 25th epoch, the training and validation accuracy values are almost equal, and then, the network starts to slightly overfit the training data.

Reducing overfitting

One way to decrease the effect of overfitting is to increase the regularization strength via L2 regularization, which we introduced in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*. Another useful technique for tackling overfitting in NNs is dropout, which will be covered in *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*.

Finally, let's evaluate the generalization performance of the model by calculating the prediction accuracy on the test dataset:

```
>>> test_mse, test_acc = compute_mse_and_acc(model, X_test, y_test)
>>> print(f'Test accuracy: {test_acc*100:.2f}%')
Test accuracy: 94.51%
```

We can see that the test accuracy is very close to the validation set accuracy corresponding to the last epoch (94.74%), which we reported during the training in the last subsection. Moreover, the respective training accuracy is only minimally higher at 95.59%, reaffirming that our model only slightly overfits the training data.

To further fine-tune the model, we could change the number of hidden units, the learning rate, or use various other tricks that have been developed over the years but are beyond the scope of this book. In *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*, you will learn about a different NN architecture that is known for its good performance on image datasets.

Also, the chapter will introduce additional performance-enhancing tricks such as adaptive learning rates, more sophisticated SGD-based optimization algorithms, batch normalization, and dropout.

Other common tricks that are beyond the scope of the following chapters include:

- Adding skip-connections, which are the main contribution of residual NNs (*Deep residual learning for image recognition* by *K. He*, *X. Zhang*, *S. Ren*, and *J. Sun*, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770-778, 2016)
- Using learning rate schedulers that change the learning rate during training (*Cyclical learning rates for training neural networks* by *L.N. Smith*, *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464-472, 2017)
- Attaching loss functions to earlier layers in the networks as it's being done in the popular Inception v3 architecture (*Rethinking the Inception architecture for computer vision by C. Szegedy*, *V. Vanhoucke*, *S. Ioffe*, *J. Shlens*, and *Z. Wojna*, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818-2826, 2016)

Lastly, let's take a look at some of the images that our MLP struggles with by extracting and plotting the first 25 misclassified samples from the test set:

```
>>> X_test_subset = X_test[:1000, :]
>>> y_test_subset = y_test[:1000]
>>> _, probas = model.forward(X_test_subset)
>>> test_pred = np.argmax(probas, axis=1)
>>> misclassified_images = \
...      X_test_subset[y_test_subset != test_pred][:25]
>>> misclassified_labels = test_pred[y_test_subset != test_pred][:25]
>>> correct_labels = y_test_subset[y_test_subset != test_pred][:25]

>>> fig, ax = plt.subplots(nrows=5, ncols=5,
...                        sharex=True, sharey=True,
...                        figsize=(8, 8))
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = misclassified_images[i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
...     ax[i].set_title(f'{i+1}) '
...                     f'True: {correct_labels[i]}\n'
...                     f' Predicted: {misclassified_labels[i]}')

>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

We should now see a 5×5 subplot matrix where the first number in the subtitles indicates the plot index, the second number represents the true class label (`True`), and the third number stands for the predicted class label (`Predicted`):

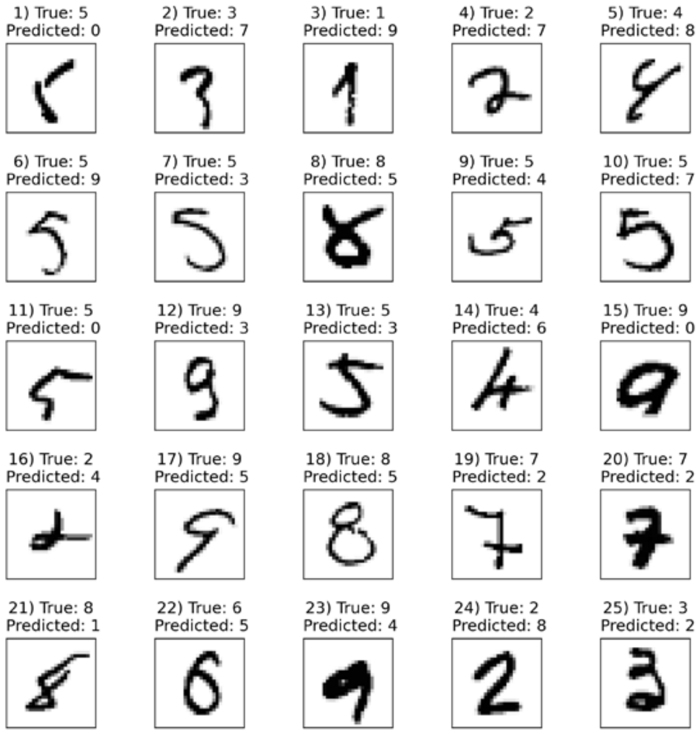

Figure 11.9: Handwritten digits that the model fails to classify correctly

As we can see in *Figure 11.9*, among others, the network finds 7s challenging when they include a horizontal line as in examples 19 and 20. Looking back at an earlier figure in this chapter where we plotted different training examples of the number 7, we can hypothesize that the handwritten digit 7 with a horizontal line is underrepresented in our dataset and is often misclassified.

Training an artificial neural network

Now that we have seen an NN in action and have gained a basic understanding of how it works by looking over the code, let's dig a little bit deeper into some of the concepts, such as the loss computation and the backpropagation algorithm that we implemented to learn the model parameters.

Computing the loss function

As mentioned previously, we used an MSE loss (as in Adaline) to train the multilayer NN as it makes the derivation of the gradients a bit easier to follow. In later chapters, we will discuss other loss functions, such as the multi-category cross-entropy loss (a generalization of the binary logistic regression loss), which is a more common choice for training NN classifiers.

In the previous section, we implemented an MLP for multiclass classification that returns an output vector of t elements that we need to compare to the t×1 dimensional target vector in the one-hot encoding representation. If we predict the class label of an input image with class label 2, using this MLP, the activation of the third layer and the target may look like this:

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Thus, our MSE loss either has to sum or average over the t activation units in our network in addition to averaging over the n examples in the dataset or mini-batch:

$$L(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{n} \sum_{1}^{n} \frac{1}{t} \sum_{j=1}^{t} \left(y_j^{[i]} - a_j^{(out)[i]} \right)^2$$

Here, again, the superscript [i] is the index of a particular example in our training dataset.

Remember that our goal is to minimize the loss function $L(\boldsymbol{W})$; thus, we need to calculate the partial derivative of the parameters $\boldsymbol{W}$ with respect to each weight for every layer in the network:

$$\frac{\partial}{\partial w_{j,l}^{(l)}} = L(\boldsymbol{W}, \boldsymbol{b})$$

In the next section, we will talk about the backpropagation algorithm, which allows us to calculate those partial derivatives to minimize the loss function.

Note that $\boldsymbol{W}$ consists of multiple matrices. In an MLP with one hidden layer, we have the weight matrix, $\boldsymbol{W}^{(h)}$, which connects the input to the hidden layer, and $\boldsymbol{W}^{(out)}$, which connects the hidden layer to the output layer. A visualization of the three-dimensional tensor $\boldsymbol{W}$ is provided in *Figure 11.10*:

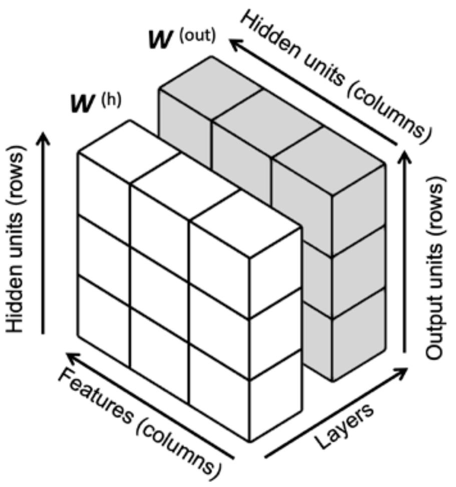

Figure 11.10: A visualization of a three-dimensional tensor

In this simplified figure, it may seem that both $\boldsymbol{W}^{(h)}$ and $\boldsymbol{W}^{(out)}$ have the same number of rows and columns, which is typically not the case unless we initialize an MLP with the same number of hidden units, output units, and input features.

If this sounds confusing, stay tuned for the next section, where we will discuss the dimensionality of $\boldsymbol{W}^{(h)}$ and $\boldsymbol{W}^{(out)}$ in more detail in the context of the backpropagation algorithm. Also, you are encouraged to read through the code of `NeuralNetMLP` again, which is annotated with helpful comments about the dimensionality of the different matrices and vector transformations.

Developing your understanding of backpropagation

Although backpropagation was introduced to the neural network community more than 30 years ago (*Learning representations by backpropagating errors*, by *D.E. Rumelhart*, *G.E. Hinton*, and *R.J. Williams*, *Nature*, 323: 6088, pages 533–536, 1986), it remains one of the most widely used algorithms for training artificial NNs very efficiently. If you are interested in additional references regarding the history of backpropagation, Juergen Schmidhuber wrote a nice survey article, *Who Invented Backpropagation?*, which you can find online at `http://people.idsia.ch/~juergen/who-invented-backpropagation.html`.

This section will provide both a short, clear summary and the bigger picture of how this fascinating algorithm works before we dive into more mathematical details. In essence, we can think of backpropagation as a very computationally efficient approach to compute the partial derivatives of a complex, non-convex loss function in multilayer NNs. Here, our goal is to use those derivatives to learn the weight coefficients for parameterizing such a multilayer artificial NN. The challenge in the parameterization of NNs is that we are typically dealing with a very large number of model parameters in a high-dimensional feature space. In contrast to loss functions of single-layer NNs such as Adaline or logistic regression, which we have seen in previous chapters, the error surface of an NN loss function is not convex or smooth with respect to the parameters. There are many bumps in this high-dimensional loss surface (local minima) that we have to overcome in order to find the global minimum of the loss function.

You may recall the concept of the chain rule from your introductory calculus classes. The chain rule is an approach to compute the derivative of a complex, nested function, such as $f(g(x))$, as follows:

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Similarly, we can use the chain rule for an arbitrarily long function composition. For example, let's assume that we have five different functions, $f(x)$, $g(x)$, $h(x)$, $u(x)$, and $v(x)$, and let F be the function composition: $F(x) = f(g(h(u(v(x)))))$. Applying the chain rule, we can compute the derivative of this function as follows:

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

In the context of computer algebra, a set of techniques, known as **automatic differentiation**, has been developed to solve such problems very efficiently. If you are interested in learning more about automatic differentiation in machine learning applications, read A.G. Baydin and B.A. Pearlmutter's article, *Automatic Differentiation of Algorithms for Machine Learning*, arXiv preprint arXiv:1404.7456, 2014, which is freely available on arXiv at `http://arxiv.org/pdf/1404.7456.pdf`.

Automatic differentiation comes with two modes, the forward and reverse modes; backpropagation is simply a special case of reverse-mode automatic differentiation. The key point is that applying the chain rule in forward mode could be quite expensive since we would have to multiply large matrices for each layer (Jacobians) that we would eventually multiply by a vector to obtain the output.

The trick of reverse mode is that we traverse the chain rule from right to left. We multiply a matrix by a vector, which yields another vector that is multiplied by the next matrix, and so on. Matrix-vector multiplication is computationally much cheaper than matrix-matrix multiplication, which is why backpropagation is one of the most popular algorithms used in NN training.

A basic calculus refresher

To fully understand backpropagation, we need to borrow certain concepts from differential calculus, which is outside the scope of this book. However, you can refer to a review chapter of the most fundamental concepts, which you might find useful in this context. It discusses function derivatives, partial derivatives, gradients, and the Jacobian. This text is freely accessible at `https://sebastianraschka.com/pdf/books/dlb/appendix_d_calculus.pdf`. If you are unfamiliar with calculus or need a brief refresher, consider reading this text as an additional supporting resource before reading the next section.

Training neural networks via backpropagation

In this section, we will go through the math of backpropagation to understand how you can learn the weights in an NN very efficiently. Depending on how comfortable you are with mathematical representations, the following equations may seem relatively complicated at first.

In a previous section, we saw how to calculate the loss as the difference between the activation of the last layer and the target class label. Now, we will see how the backpropagation algorithm works to update the weights in our MLP model from a mathematical perspective, which we implemented in the `.backward()` method of the `NeuralNetMLP()` class. As we recall from the beginning of this chapter, we first need to apply forward propagation to obtain the activation of the output layer, which we formulated as follows:

$$\boldsymbol{Z}^{(h)} = \boldsymbol{X}^{(in)}\boldsymbol{W}^{(h)T} + \boldsymbol{b}^{(h)} \qquad \text{(net input of the hidden layer)}$$

$$\boldsymbol{A}^{(h)} = \sigma\left(\boldsymbol{Z}^{(h)}\right) \qquad \text{(activation of the hidden layer)}$$

$$\boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)T} + \boldsymbol{b}^{(out)} \qquad \text{(net input of the output layer)}$$

$$\boldsymbol{A}^{(out)} = \sigma\left(\boldsymbol{Z}^{(out)}\right) \qquad \text{(activation of the output layer)}$$

Concisely, we just forward-propagate the input features through the connections in the network, as shown by the arrows in *Figure 11.11* for a network with two input features, three hidden nodes, and two output nodes:

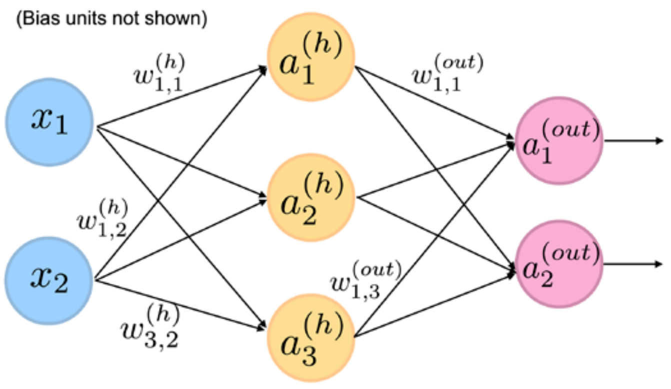

Figure 11.11: Forward-propagating the input features of an NN

In backpropagation, we propagate the error from right to left. We can think of this as an application of the chain rule to the computation of the forward pass to compute the gradient of the loss with respect to the model weights (and bias units). For simplicity, we will illustrate this process for the partial derivative used to update the first weight in the weight matrix of the output layer. The paths of the computation we backpropagate are highlighted via the bold arrows below:

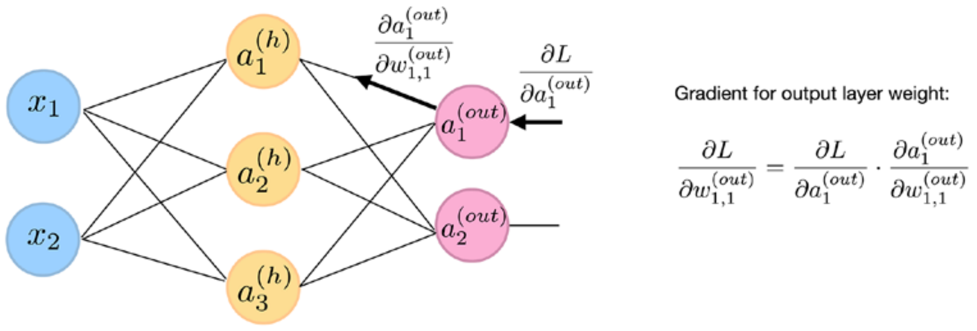
$\frac{\partial a_1^{(out)}}{\partial w_{1,1}^{(out)}}$
$\frac{\partial L}{\partial a_1^{(out)}}$
Gradient for output layer weight:
$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial w_{1,1}^{(out)}}$
x_1
x_2
$a_1^{(h)}$
$a_2^{(h)}$
$a_3^{(h)}$
$a_1^{(out)}$
$a_2^{(out)}$

Figure 11.12: Backpropagating the error of an NN

If we include the net inputs *z* explicitly, the partial derivative computation shown in the previous figure expands as follows:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

To compute this partial derivative, which is used to update $w_{1,1}^{(out)}$, we can compute the three individual partial derivative terms and multiply the results. For simplicity, we will omit averaging over the individual examples in the mini-batch, so we drop the $\frac{1}{n}\sum_{i=1}^{n}$ averaging term from the following equations.

Let's start with $\frac{\partial L}{\partial a_1^{(out)}}$, which is the partial derivative of the MSE loss (which simplifies to the squared error if we omit the mini-batch dimension) with respect to the predicted output score of the first output node:

$$\frac{\partial L}{\partial a_1^{(out)}} = \frac{\partial}{\partial a_1^{(out)}}\left(y_1 - a_1^{(out)}\right)^2 = 2\left(a_1^{(out)} - y\right)$$

The next term is the derivative of the logistic sigmoid activation function that we used in the output layer:

$$\frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} = \frac{\partial}{\partial z_1^{(out)}}\frac{1}{1+e^{z_1^{(out)}}} = \quad \ldots \quad = \left(\frac{1}{1+e^{z_1^{(out)}}}\right)\left(1 - \frac{1}{1+e^{z_1^{(out)}}}\right)$$

$$= a_1^{(out)}\left(1 - a_1^{(out)}\right)$$

Lastly, we compute the derivative of the net input with respect to the weight:

$$\frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = \frac{\partial}{\partial w_{1,1}^{(out)}} a_1^{(h)} w_{1,1}^{(out)} + b_1^{(out)} = a_1^{(h)}$$

Putting all of it together, we get the following:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = 2\left(a_1^{(out)} - y\right) \cdot a_1^{(out)}\left(1 - a_1^{(out)}\right) \cdot a_1^{(h)}$$

We then use this value to update the weight via the familiar stochastic gradient descent update with a learning rate of η:

$$w_{1,1}^{(out)} := w_{1,1}^{(out)} - \eta \frac{\partial L}{\partial w_{1,1}^{(out)}}$$

In our code implementation of `NeuralNetMLP()`, we implemented the computation $\frac{\partial L}{\partial w_{1,1}^{(out)}}$ in vectorized form in the `.backward()` method as follows:

```
        # Part 1: dLoss/dOutWeights
        ## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
        ## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet for convenient re-use

        # input/output dim: [n_examples, n_classes]
        d_loss__d_a_out = 2.*(a_out - y_onehot) / y.shape[0]

        # input/output dim: [n_examples, n_classes]
        d_a_out__d_z_out = a_out * (1. - a_out) # sigmoid derivative

        # output dim: [n_examples, n_classes]
        delta_out = d_loss__d_a_out * d_a_out__d_z_out # "delta (rule)
                                                       # placeholder"

        # gradient for output weights

        # [n_examples, n_hidden]
        d_z_out__dw_out = a_h

        # input dim: [n_classes, n_examples] dot [n_examples, n_hidden]
        # output dim: [n_classes, n_hidden]
        d_loss__dw_out = np.dot(delta_out.T, d_z_out__dw_out)
        d_loss__db_out = np.sum(delta_out, axis=0)
```

As annotated in the code snippet above, we created the following "delta" placeholder variable:

$$\delta_1^{(out)} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}}$$

This is because $\delta^{(out)}$ terms are involved in computing the partial derivatives (or gradients) of the hidden layer weights as well; hence, we can reuse $\delta^{(out)}$.

Speaking of hidden layer weights, *Figure 11.13* illustrates how to compute the partial derivative of the loss with respect to the first weight of the hidden layer:

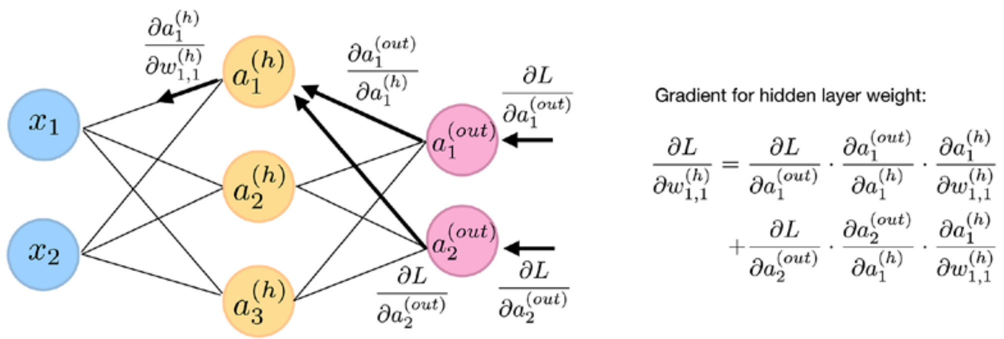
x_1
x_2
$a_1^{(h)}$
$a_2^{(h)}$
$a_3^{(h)}$
$a_1^{(out)}$
$a_2^{(out)}$
$\frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}}$
$\frac{\partial a_1^{(out)}}{\partial a_1^{(h)}}$
$\frac{\partial L}{\partial a_1^{(out)}}$
$\frac{\partial L}{\partial a_2^{(out)}}$
$\frac{\partial L}{\partial a_2^{(out)}}$
Gradient for hidden layer weight:
$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}} + \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}}$

Figure 11.13: Computing the partial derivatives of the loss with respect to the first hidden layer weight

It is important to highlight that since the weight $w_{1,1}^{(h)}$ is connected to both output nodes, we have to use the *multi-variable* chain rule to sum the two paths highlighted with bold arrows. As before, we can expand it to include the net inputs z and then solve the individual terms:

$$\begin{aligned}\frac{\partial L}{\partial w_{1,1}^{(out)}} &= \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} \\ &+ \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial z_2^{(out)}} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}\end{aligned}$$

Notice that if we reuse $\delta^{(out)}$ computed previously, this equation can be simplified as follows:

$$\begin{aligned}\frac{\partial L}{\partial w_{1,1}^{(out)}} &= \delta_1^{(out)} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} \\ &+ \delta_2^{(out)} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}\end{aligned}$$

The preceding terms can be individually solved relatively easily, as we have done previously, because there are no new derivatives involved. For example, $\frac{\partial a_1^{(h)}}{\partial z_1^{(h)}}$ is the derivative of the sigmoid activation, that is, $a_1^{(h)}\left(1 - a_1^{(h)}\right)$, and so forth. We'll leave solving the individual parts as an optional exercise for you.

About convergence in neural networks

You might be wondering why we did not use regular gradient descent but instead used mini-batch learning to train our NN for the handwritten digit classification earlier. You may recall our discussion on SGD that we used to implement online learning. In online learning, we compute the gradient based on a single training example ($k = 1$) at a time to perform the weight update. Although this is a stochastic approach, it often leads to very accurate solutions with a much faster convergence than regular gradient descent. Mini-batch learning is a special form of SGD where we compute the gradient based on a subset k of the n training examples with $1 < k < n$. Mini-batch learning has an advantage over online learning in that we can make use of our vectorized implementations to improve computational efficiency. However, we can update the weights much faster than in regular gradient descent. Intuitively, you can think of mini-batch learning as predicting the voter turnout of a presidential election from a poll by asking only a representative subset of the population rather than asking the entire population (which would be equal to running the actual election).

Multilayer NNs are much harder to train than simpler algorithms such as Adaline, logistic regression, or support vector machines. In multilayer NNs, we typically have hundreds, thousands, or even billions of weights that we need to optimize. Unfortunately, the output function has a rough surface, and the optimization algorithm can easily become trapped in local minima, as shown in *Figure 11.14*:

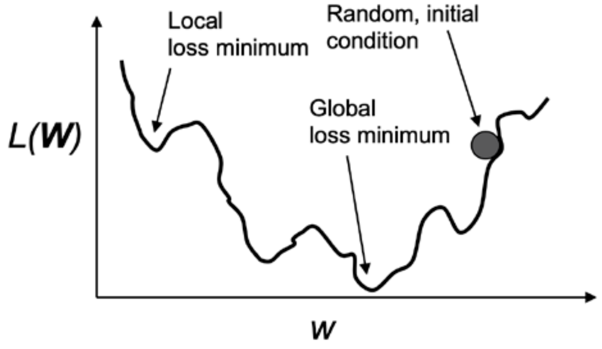

Figure 11.14: Optimization algorithms can become trapped in local minima

Note that this representation is extremely simplified since our NN has many dimensions; it makes it impossible to visualize the actual loss surface for the human eye. Here, we only show the loss surface for a single weight on the x axis. However, the main message is that we do not want our algorithm to get trapped in local minima. By increasing the learning rate, we can more readily escape such local minima. On the other hand, we also increase the chance of overshooting the global optimum if the learning rate is too large. Since we initialize the weights randomly, we start with a solution to the optimization problem that is typically hopelessly wrong.

A few last words about the neural network implementation

You may be wondering why we went through all of this theory just to implement a simple multilayer artificial network that can classify handwritten digits instead of using an open source Python machine learning library. In fact, we will introduce more complex NN models in the next chapters, which we will train using the open source PyTorch library (`https://pytorch.org`).

Although the from-scratch implementation in this chapter seems a bit tedious at first, it was a good exercise for understanding the basics behind backpropagation and NN training. A basic understanding of algorithms is crucial for applying machine learning techniques appropriately and successfully.

Now that you have learned how feedforward NNs work, we are ready to explore more sophisticated DNNs using PyTorch, which allows us to construct NNs more efficiently, as we will see in *Chapter 12, Parallelizing Neural Network Training with PyTorch*.

PyTorch, which was originally released in September 2016, has gained a lot of popularity among machine learning researchers, who use it to construct DNNs because of its ability to optimize mathematical expressions for computations on multidimensional arrays utilizing **graphics processing units** (**GPUs**).

Lastly, we should note that scikit-learn also includes a basic MLP implementation, `MLPClassifier`, which you can find at `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`. While this implementation is great and very convenient for training basic MLPs, we strongly recommend specialized deep learning libraries, such as PyTorch, for implementing and training multilayer NNs.

Summary

In this chapter, you have learned the basic concepts behind multilayer artificial NNs, which are currently the hottest topic in machine learning research. In *Chapter 2, Training Simple Machine Learning Algorithms for Classification*, we started our journey with simple single-layer NN structures and now we have connected multiple neurons to a powerful NN architecture to solve complex problems such as handwritten digit recognition. We demystified the popular backpropagation algorithm, which is one of the building blocks of many NN models that are used in deep learning. After learning about the backpropagation algorithm in this chapter, we are well equipped for exploring more complex DNN architectures. In the remaining chapters, we will cover more advanced deep learning concepts and PyTorch, an open source library that allows us to implement and train multilayer NNs more efficiently.

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:
`https://packt.link/MLwPyTorch`