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*:

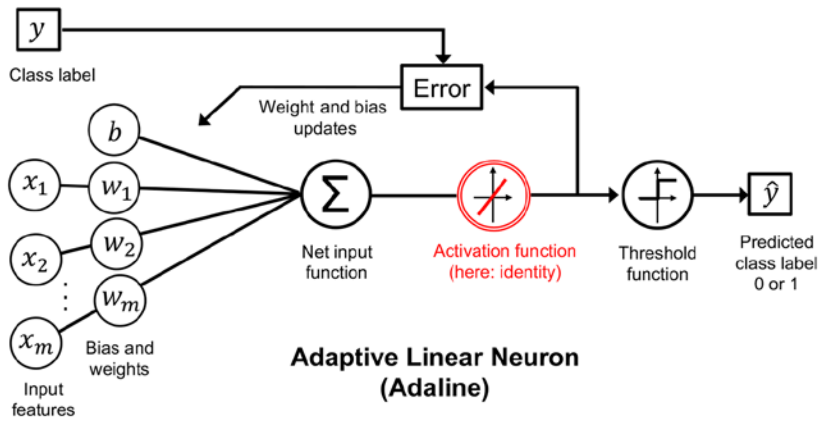

Figure 11.1: The Adaline algorithm

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:

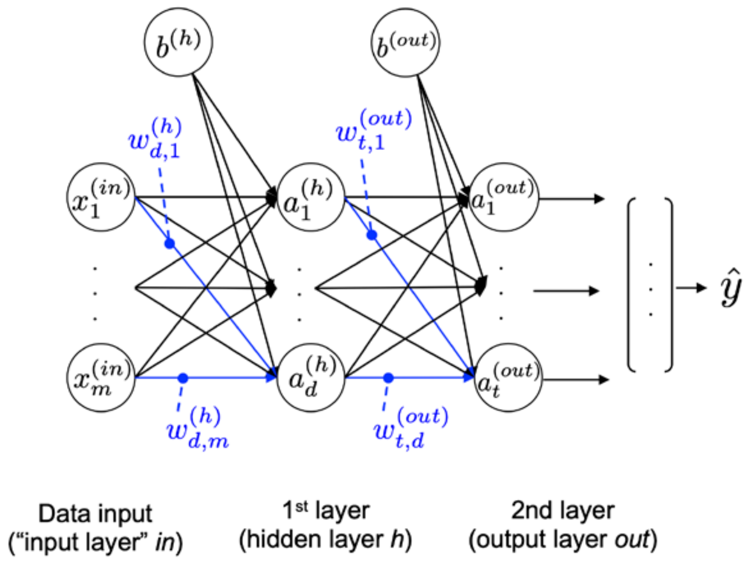

Figure 11.2: A two-layer MLP

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*:

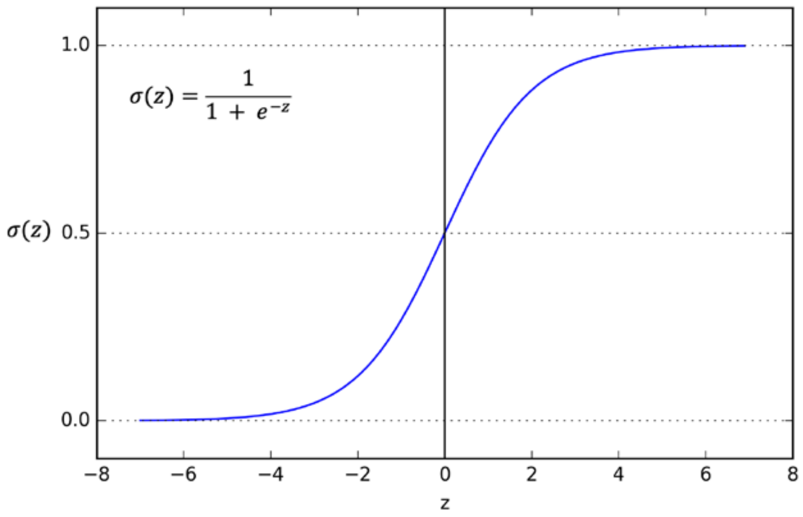

Figure 11.3: The sigmoid activation function

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.