

Chapter 6

Deep Feedforward Networks

Deep feedforward networks, also often called **feedforward neural networks**, or **multilayer perceptrons** (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f^* . For example, for a classifier, $y = f^*(\mathbf{x})$ maps an input \mathbf{x} to a category y . A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.

These models are called **feedforward** because information flows through the function being evaluated from \mathbf{x} , through the intermediate computations used to define f , and finally to the output \mathbf{y} . There are no **feedback** connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks**, presented in chapter 10.

Feedforward networks are of extreme importance to machine learning practitioners. They form the basis of many important commercial applications. For example, the convolutional networks used for object recognition from photos are a specialized kind of feedforward network. Feedforward networks are a conceptual stepping stone on the path to recurrent networks, which power many natural language applications.

Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the **first layer** of the network, $f^{(2)}$ is called the **second layer**, and so on. The overall

length of the chain gives the **depth** of the model. It is from this terminology that the name “deep learning” arises. The final layer of a feedforward network is called the **output layer**. During neural network training, we drive $f(\mathbf{x})$ to match $f^*(\mathbf{x})$. The training data provides us with noisy, approximate examples of $f^*(\mathbf{x})$ evaluated at different training points. Each example \mathbf{x} is accompanied by a label $y \approx f^*(\mathbf{x})$. The training examples specify directly what the output layer must do at each point \mathbf{x} ; it must produce a value that is close to y . The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f^* . Because the training data does not show the desired output for each of these layers, these layers are called **hidden layers**.

Finally, these networks are called *neural* because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the **width** of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many **units** that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value. The idea of using many layers of vector-valued representation is drawn from neuroscience. The choice of the functions $f^{(i)}(\mathbf{x})$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute. However, modern neural network research is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they may be fit efficiently and reliably, either in closed form or with convex optimization. Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

To extend linear models to represent nonlinear functions of \mathbf{x} , we can apply the linear model not to \mathbf{x} itself but to a transformed input $\phi(\mathbf{x})$, where ϕ is a

nonlinear transformation. Equivalently, we can apply the kernel trick described in section 5.7.2, to obtain a nonlinear learning algorithm based on implicitly applying the ϕ mapping. We can think of ϕ as providing a set of features describing \mathbf{x} , or as providing a new representation for \mathbf{x} .

The question is then how to choose the mapping ϕ .

1. One option is to use a very generic ϕ , such as the infinite-dimensional ϕ that is implicitly used by kernel machines based on the RBF kernel. If $\phi(\mathbf{x})$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor. Very generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.
2. Another option is to manually engineer ϕ . Until the advent of deep learning, this was the dominant approach. This approach requires decades of human effort for each separate task, with practitioners specializing in different domains such as speech recognition or computer vision, and with little transfer between domains.
3. The strategy of deep learning is to learn ϕ . In this approach, we have a model $y = f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{w}) = \phi(\mathbf{x}; \boldsymbol{\theta})^\top \mathbf{w}$. We now have parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of functions, and parameters \mathbf{w} that map from $\phi(\mathbf{x})$ to the desired output. This is an example of a deep feedforward network, with ϕ defining a hidden layer. This approach is the only one of the three that gives up on the convexity of the training problem, but the benefits outweigh the harms. In this approach, we parametrize the representation as $\phi(\mathbf{x}; \boldsymbol{\theta})$ and use the optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation. If we wish, this approach can capture the benefit of the first approach by being highly generic—we do so by using a very broad family $\phi(\mathbf{x}; \boldsymbol{\theta})$. This approach can also capture the benefit of the second approach. Human practitioners can encode their knowledge to help generalization by designing families $\phi(\mathbf{x}; \boldsymbol{\theta})$ that they expect will perform well. The advantage is that the human designer only needs to find the right general function family rather than finding precisely the right function.

This general principle of improving models by learning features extends beyond the feedforward networks described in this chapter. It is a recurring theme of deep learning that applies to all of the kinds of models described throughout this book. Feedforward networks are the application of this principle to learning deterministic

mappings from \mathbf{x} to \mathbf{y} that lack feedback connections. Other models presented later will apply these principles to learning stochastic mappings, learning functions with feedback, and learning probability distributions over a single vector.

We begin this chapter with a simple example of a feedforward network. Next, we address each of the design decisions needed to deploy a feedforward network. First, training a feedforward network requires making many of the same design decisions as are necessary for a linear model: choosing the optimizer, the cost function, and the form of the output units. We review these basics of gradient-based learning, then proceed to confront some of the design decisions that are unique to feedforward networks. Feedforward networks have introduced the concept of a hidden layer, and this requires us to choose the **activation functions** that will be used to compute the hidden layer values. We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should be in each layer. Learning in deep neural networks requires computing the gradients of complicated functions. We present the **back-propagation** algorithm and its modern generalizations, which can be used to efficiently compute these gradients. Finally, we close with some historical perspective.

6.1 Example: Learning XOR

To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function.

The XOR function (“exclusive or”) is an operation on two binary values, x_1 and x_2 . When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0. The XOR function provides the target function $y = f^*(\mathbf{x})$ that we want to learn. Our model provides a function $y = f(\mathbf{x}; \boldsymbol{\theta})$ and our learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^* .

In this simple example, we will not be concerned with statistical generalization. We want our network to perform correctly on the four points $\mathbb{X} = \{[0, 0]^\top, [0, 1]^\top, [1, 0]^\top, \text{ and } [1, 1]^\top\}$. We will train the network on all four of these points. The only challenge is to fit the training set.

We can treat this problem as a regression problem and use a mean squared error loss function. We choose this loss function to simplify the math for this example as much as possible. In practical applications, MSE is usually not an

appropriate cost function for modeling binary data. More appropriate approaches are described in section 6.2.2.2.

Evaluated on our whole training set, the MSE loss function is

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\mathbf{x} \in \mathbb{X}} (f^*(\mathbf{x}) - f(\mathbf{x}; \boldsymbol{\theta}))^2. \quad (6.1)$$

Now we must choose the form of our model, $f(\mathbf{x}; \boldsymbol{\theta})$. Suppose that we choose a linear model, with $\boldsymbol{\theta}$ consisting of \mathbf{w} and b . Our model is defined to be

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^\top \mathbf{w} + b. \quad (6.2)$$

We can minimize $J(\boldsymbol{\theta})$ in closed form with respect to \mathbf{w} and b using the normal equations.

After solving the normal equations, we obtain $\mathbf{w} = \mathbf{0}$ and $b = \frac{1}{2}$. The linear model simply outputs 0.5 everywhere. Why does this happen? Figure 6.1 shows how a linear model is not able to represent the XOR function. One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.

Specifically, we will introduce a very simple feedforward network with one hidden layer containing two hidden units. See figure 6.2 for an illustration of this model. This feedforward network has a vector of hidden units \mathbf{h} that are computed by a function $f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$. The values of these hidden units are then used as the input for a second layer. The second layer is the output layer of the network. The output layer is still just a linear regression model, but now it is applied to \mathbf{h} rather than to \mathbf{x} . The network now contains two functions chained together: $\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$ and $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$, with the complete model being $f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$.

What function should $f^{(1)}$ compute? Linear models have served us well so far, and it may be tempting to make $f^{(1)}$ be linear as well. Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input. Ignoring the intercept terms for the moment, suppose $f^{(1)}(\mathbf{x}) = \mathbf{W}^\top \mathbf{x}$ and $f^{(2)}(\mathbf{h}) = \mathbf{h}^\top \mathbf{w}$. Then $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{W}^\top \mathbf{x}$. We could represent this function as $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}'$ where $\mathbf{w}' = \mathbf{W}\mathbf{w}$.

Clearly, we must use a nonlinear function to describe the features. Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an activation function. We use that strategy here, by defining $\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{c})$, where \mathbf{W} provides the weights of a linear transformation and \mathbf{c} the biases. Previously, to describe a linear regression

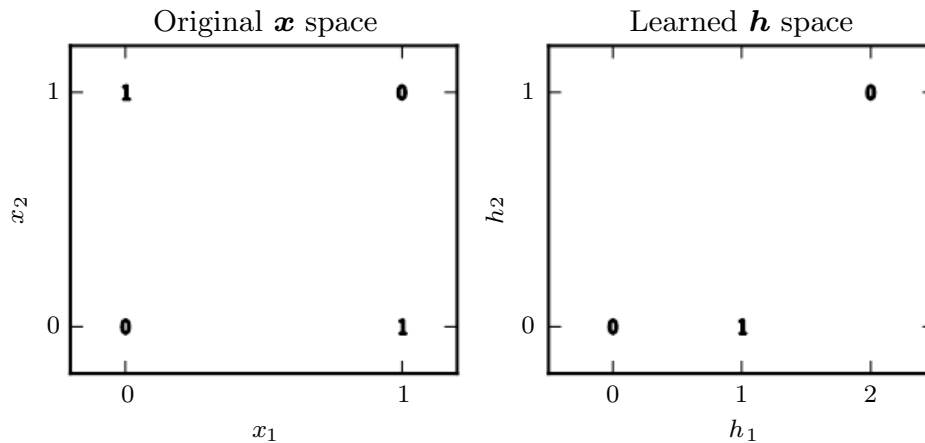


Figure 6.1: Solving the XOR problem by learning a representation. The bold numbers printed on the plot indicate the value that the learned function must output at each point. *(Left)* A linear model applied directly to the original input cannot implement the XOR function. When $x_1 = 0$, the model's output must increase as x_2 increases. When $x_1 = 1$, the model's output must decrease as x_2 increases. A linear model must apply a fixed coefficient w_2 to x_2 . The linear model therefore cannot use the value of x_1 to change the coefficient on x_2 and cannot solve this problem. *(Right)* In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem. In our example solution, the two points that must have output 1 have been collapsed into a single point in feature space. In other words, the nonlinear features have mapped both $\mathbf{x} = [1, 0]^\top$ and $\mathbf{x} = [0, 1]^\top$ to a single point in feature space, $\mathbf{h} = [1, 0]^\top$. The linear model can now describe the function as increasing in h_1 and decreasing in h_2 . In this example, the motivation for learning the feature space is only to make the model capacity greater so that it can fit the training set. In more realistic applications, learned representations can also help the model to generalize.

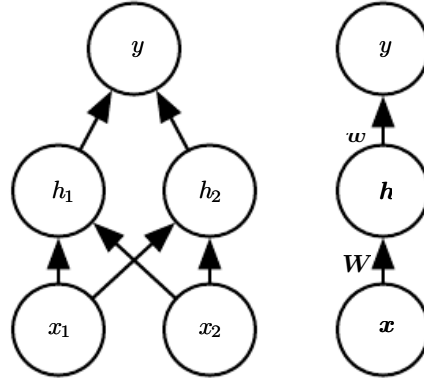


Figure 6.2: An example of a feedforward network, drawn in two different styles. Specifically, this is the feedforward network we use to solve the XOR example. It has a single hidden layer containing two units. *(Left)* In this style, we draw every unit as a node in the graph. This style is very explicit and unambiguous but for networks larger than this example it can consume too much space. *(Right)* In this style, we draw a node in the graph for each entire vector representing a layer’s activations. This style is much more compact. Sometimes we annotate the edges in this graph with the name of the parameters that describe the relationship between two layers. Here, we indicate that a matrix \mathbf{W} describes the mapping from \mathbf{x} to \mathbf{h} , and a vector \mathbf{w} describes the mapping from \mathbf{h} to y . We typically omit the intercept parameters associated with each layer when labeling this kind of drawing.

model, we used a vector of weights and a scalar bias parameter to describe an affine transformation from an input vector to an output scalar. Now, we describe an affine transformation from a vector \mathbf{x} to a vector \mathbf{h} , so an entire vector of bias parameters is needed. The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\mathbf{x}^\top \mathbf{W}_{:,i} + c_i)$. In modern neural networks, the default recommendation is to use the **rectified linear unit** or ReLU (Jarrett *et al.*, 2009; Nair and Hinton, 2010; Glorot *et al.*, 2011a) defined by the activation function $g(z) = \max\{0, z\}$ depicted in figure 6.3.

We can now specify our complete network as

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^\top \max\{0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}\} + b. \quad (6.3)$$

We can now specify a solution to the XOR problem. Let

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad (6.4)$$

$$\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad (6.5)$$

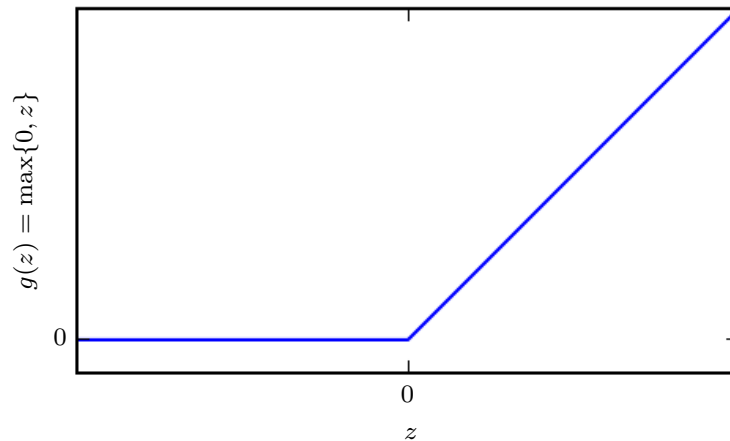


Figure 6.3: The rectified linear activation function. This activation function is the default activation function recommended for use with most feedforward neural networks. Applying this function to the output of a linear transformation yields a nonlinear transformation. However, the function remains very close to linear, in the sense that it is a piecewise linear function with two linear pieces. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods. They also preserve many of the properties that make linear models generalize well. A common principle throughout computer science is that we can build complicated systems from minimal components. Much as a Turing machine’s memory needs only to be able to store 0 or 1 states, we can build a universal function approximator from rectified linear functions.

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad (6.6)$$

and $b = 0$.

We can now walk through the way that the model processes a batch of inputs. Let \mathbf{X} be the design matrix containing all four points in the binary input space, with one example per row:

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \quad (6.7)$$

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \quad (6.8)$$

Next, we add the bias vector \mathbf{c} , to obtain

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.9)$$

In this space, all of the examples lie along a line with slope 1. As we move along this line, the output needs to begin at 0, then rise to 1, then drop back down to 0. A linear model cannot implement such a function. To finish computing the value of \mathbf{h} for each example, we apply the rectified linear transformation:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.10)$$

This transformation has changed the relationship between the examples. They no longer lie on a single line. As shown in figure 6.1, they now lie in a space where a linear model can solve the problem.

We finish by multiplying by the weight vector \mathbf{w} :

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \quad (6.11)$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here. Instead, a gradient-based optimization algorithm can find parameters that produce very little error. The solution we described to the XOR problem is at a global minimum of the loss function, so gradient descent could converge to this point. There are other equivalent solutions to the XOR problem that gradient descent could also find. The convergence point of gradient descent depends on the initial values of the parameters. In practice, gradient descent would usually not find clean, easily understood, integer-valued solutions like the one we presented here.

6.2 Gradient-Based Learning

Designing and training a neural network is not much different from training any other machine learning model with gradient descent. In section 5.10, we described how to build a machine learning algorithm by specifying an optimization procedure, a cost function, and a model family.

The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Convex optimization converges starting from any initial parameters (in theory—in practice it is very robust but can encounter numerical problems). Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values. The iterative gradient-based optimization algorithms used to train feedforward networks and almost all other deep models will be described in detail in chapter 8, with parameter initialization in particular discussed in section 8.4. For the moment, it suffices to understand that the training algorithm is almost always based on using the gradient to descend the cost function in one way or another. The specific algorithms are improvements and refinements on the ideas of gradient descent, introduced in section 4.3, and,

more specifically, are most often improvements of the stochastic gradient descent algorithm, introduced in section 5.9.

We can of course, train models such as linear regression and support vector machines with gradient descent too, and in fact this is common when the training set is extremely large. From this point of view, training a neural network is not much different from training any other model. Computing the gradient is slightly more complicated for a neural network, but can still be done efficiently and exactly. Section 6.5 will describe how to obtain the gradient using the back-propagation algorithm and modern generalizations of the back-propagation algorithm.

As with other machine learning models, to apply gradient-based learning we must choose a cost function, and we must choose how to represent the output of the model. We now revisit these design considerations with special emphasis on the neural networks scenario.

6.2.1 Cost Functions

An important aspect of the design of a deep neural network is the choice of the cost function. Fortunately, the cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.

In most cases, our parametric model defines a distribution $p(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ and we simply use the principle of maximum likelihood. This means we use the cross-entropy between the training data and the model's predictions as the cost function.

Sometimes, we take a simpler approach, where rather than predicting a complete probability distribution over \mathbf{y} , we merely predict some statistic of \mathbf{y} conditioned on \mathbf{x} . Specialized loss functions allow us to train a predictor of these estimates.

The total cost function used to train a neural network will often combine one of the primary cost functions described here with a regularization term. We have already seen some simple examples of regularization applied to linear models in section 5.2.2. The weight decay approach used for linear models is also directly applicable to deep neural networks and is among the most popular regularization strategies. More advanced regularization strategies for neural networks will be described in chapter 7.

6.2.1.1 Learning Conditional Distributions with Maximum Likelihood

Most modern neural networks are trained using maximum likelihood. This means that the cost function is simply the negative log-likelihood, equivalently described