Chapter 6

Deep Feedforward Networks

Deep feedforward networks, also 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, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ 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 $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. 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**, as 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 $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. 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. The name "deep learning" arose from this terminology. The final layer of a feedforward network is called the **output layer**. During neural network training, we drive $f(\boldsymbol{x})$ to match $f^*(\boldsymbol{x})$. The training data provides us with noisy, approximate examples of $f^*(\boldsymbol{x})$ evaluated at different training points. Each example $\boldsymbol{x}$ is accompanied by a label $y \approx f^*(\boldsymbol{x})$. The training examples specify directly what the output layer must do at each point $\boldsymbol{x}$; 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 do 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, they 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 representations is drawn from neuroscience. The choice of the functions $f^{(i)}(\boldsymbol{x})$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute. Modern neural network research, however, 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 can 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 $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$, 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 $\boldsymbol{x}$, or as providing a new representation for $\boldsymbol{x}$.

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 $\phi(\boldsymbol{x})$ 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. It 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 $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}$. We now have parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of functions, and parameters $\boldsymbol{w}$ that map from $\phi(\boldsymbol{x})$ 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 $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ 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 $\phi(\boldsymbol{x}; \boldsymbol{\theta})$. Deep learning can also capture the benefit of the second approach. Human practitioners can encode their knowledge to help generalization by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ 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 the kinds of models described throughout this book. Feedforward networks are the application of this principle to learning deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections. Other models, presented later, apply these principles to learning stochastic mappings, functions with feedback, and 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.

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 $y = f^*(\boldsymbol{x})$ that we want to learn. Our model provides a function $y = f(\boldsymbol{x};\boldsymbol{\theta})$, 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 $= \{[0,0]^\top$, $[0,1]^\top$, $[1,0]^\top$, 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 have chosen 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

$$J(\boldsymbol{\theta}) = \frac{1}{4} \underset{\in}{\quad} \left(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta})\right)^2 . \tag{6.1}$$

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

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

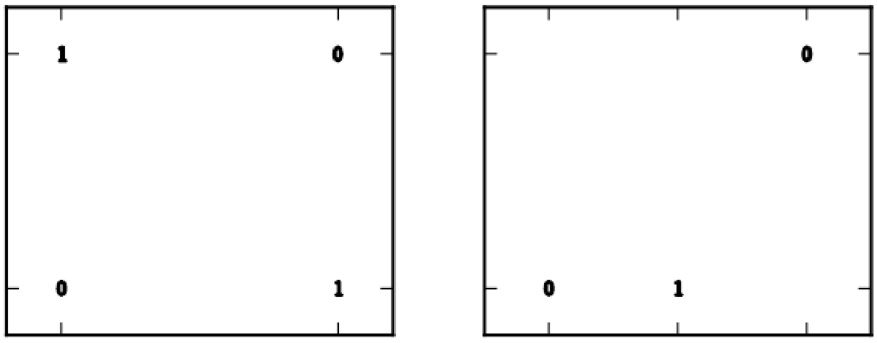
We can minimize $J(\boldsymbol{\theta})$ in closed form with respect to $\boldsymbol{w}$ and b using the normal equations.

After solving the normal equations, we obtain $\boldsymbol{w} =$ 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 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 $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c})$. 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 $\boldsymbol{h}$ rather than to $\boldsymbol{x}$. The network now contains two functions chained together, $\boldsymbol{h} = f^{(1)}(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h};\boldsymbol{w},b)$, with the complete model being $f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c},\boldsymbol{w},b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$.

What function should $f^{(1)}$ compute? Linear models have served us well so far, and it may be tempting to make $f^{(1)}$ 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 $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^\top\boldsymbol{x}$ and $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^\top\boldsymbol{w}$. Then $f(\boldsymbol{x}) = \boldsymbol{x}^\top\boldsymbol{W}\boldsymbol{w}$. We could represent this function as $f(\boldsymbol{x}) = \boldsymbol{x}^\top\boldsymbol{w}'$ where $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$.

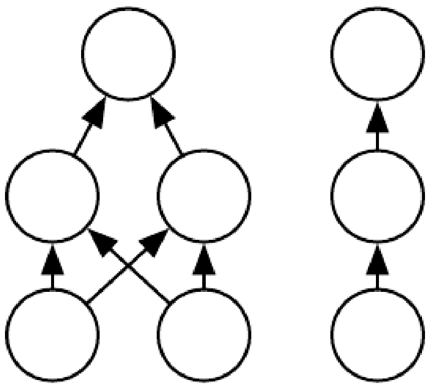
Clearly, we must use a nonlinear function to describe the features. Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed nonlinear function called an activation function. We use that strategy here, by defining $\boldsymbol{h} = g(\boldsymbol{W}^\top\boldsymbol{x}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



x $= 0$ x x $= 1$
x
w x x
x

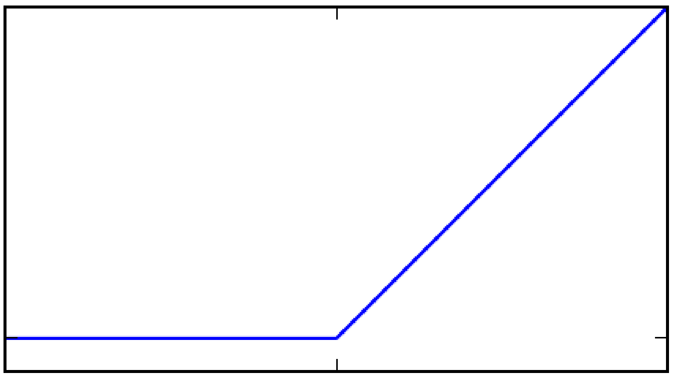
1

$\boldsymbol{x} = [1, 0]$ $\boldsymbol{x} = [0, 1]$ $\boldsymbol{h} = [1 ,0]$
h h



$\boldsymbol{W}$

$\boldsymbol{x}$ $\boldsymbol{h}$ $\boldsymbol{w}$ $\boldsymbol{h}$ y



affine transformation from an input vector to an output scalar. Now, we describe an affine transformation from a vector $\boldsymbol{x}$ to a vector $\boldsymbol{h}$, 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 $h_i = g(\boldsymbol{x}^\top \boldsymbol{W}_{:,i} + c_i)$. 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

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

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

$$\boldsymbol{W} = \begin{matrix} 1 & 1 \\ 1 & 1 \end{matrix}, \tag{6.4}$$

$$\boldsymbol{c} = \begin{matrix} 0 \\ -1 \end{matrix}, \tag{6.5}$$

$$\boldsymbol{w} = \begin{matrix} 1 \\ -2 \end{matrix}, \tag{6.6}$$

and $b = 0$.

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

$$\boldsymbol{X} = \begin{matrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{matrix}. \tag{6.7}$$

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

$$\boldsymbol{X}\boldsymbol{W} = \begin{matrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{matrix}. \tag{6.8}$$

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

$$\begin{matrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{matrix}. \tag{6.9}$$

In this space, all 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 $\boldsymbol{h}$ for each example, we apply the rectified linear transformation:

$$\begin{matrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{matrix} . \tag{6.10}$$

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 with multiplying by the weight vector $\boldsymbol{w}$:

$$\begin{matrix} 0 \\ 1 \\ 1 \\ 0 \end{matrix} . \tag{6.11}$$

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.

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 nonconvex. 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 robust but can encounter numerical problems). Stochastic gradient descent applied to nonconvex 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 are 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. In Section 6.5 we 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.

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 $p(\boldsymbol{y} \mid \boldsymbol{x};\boldsymbol{\theta})$ 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 $\boldsymbol{y}$, we merely predict some statistic of $\boldsymbol{y}$ conditioned on $\boldsymbol{x}$. Specialized loss functions enable 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 are 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 as the cross-entropy between the training data and the model distribution. This cost function is given by

$$J(\boldsymbol{\theta}) = - \quad_{,\sim\hat{p}} \quad \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}). \tag{6.12}$$

The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{\text{model}}$. The expansion of the above equation typically yields some terms that do not depend on the model parameters and may be discarded. For example, as we saw in section 5.5.1, if $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I})$, then we recover the mean squared error cost,

$$J(\theta) = \frac{1}{2} \quad_{,\sim\hat{p}} \quad ||\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})||^2 + \text{const}, \tag{6.13}$$

up to a scaling factor of $\frac{1}{2}$ and a term that does not depend on $\boldsymbol{\theta}$. The discarded constant is based on the variance of the Gaussian distribution, which in this case we chose not to parametrize. Previously, we saw that the equivalence between maximum likelihood estimation with an output distribution and minimization of mean squared error holds for a linear model, but in fact, the equivalence holds regardless of the $f(\boldsymbol{x};\boldsymbol{\theta})$ used to predict the mean of the Gaussian.

An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model. Specifying a model $p(\boldsymbol{y} \mid \boldsymbol{x})$ automatically determines a cost function $\log p(\boldsymbol{y} \mid \boldsymbol{x})$.

One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm. Functions that saturate (become very flat) undermine this objective because they make the gradient become very small. In many cases this happens because the activation functions used to produce the output of the hidden units or the output units saturate. The negative log-likelihood helps to avoid this problem for many models. Several output units involve an exp function that can saturate when its argument is very negative. The log function in the negative log-likelihood cost function undoes the exp of some output units. We will discuss the interaction between the cost function and the choice of output unit in section 6.2.2.

One unusual property of the cross-entropy cost used to perform maximum likelihood estimation is that it usually does not have a minimum value when applied to the models commonly used in practice. For discrete output variables, most models are parametrized in such a way that they cannot represent a probability of zero or one, but can come arbitrarily close to doing so. Logistic regression is an example of such a model. For real-valued output variables, if the model can control the density of the output distribution (for example, by learning the variance parameter of a Gaussian output distribution) then it becomes possible to assign extremely high density to the correct training set outputs, resulting in cross-entropy approaching negative infinity. Regularization techniques described in chapter 7 provide several different ways of modifying the learning problem so that the model cannot reap unlimited reward in this way.

6.2.1.2 Learning Conditional Statistics

Instead of learning a full probability distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, we often want to learn just one conditional statistic of $\boldsymbol{y}$ given $\boldsymbol{x}$.

For example, we may have a predictor $f(\boldsymbol{x};\boldsymbol{\theta})$ that we wish to employ to predict the mean of $\boldsymbol{y}$.

If we use a sufficiently powerful neural network, we can think of the neural network as being able to represent any function f from a wide class of functions, with this class being limited only by features such as continuity and boundedness rather than by having a specific parametric form. From this point of view, we can view the cost function as being a **functional** rather than just a function. A functional is a mapping from functions to real numbers. We can thus think of learning as choosing a function rather than merely choosing a set of parameters. We can design our cost functional to have its minimum occur at some specific function we desire. For example, we can design the cost functional to have its minimum lie on the function that maps $\boldsymbol{x}$ to the expected value of $\boldsymbol{y}$ given $\boldsymbol{x}$. Solving an optimization problem with respect to a function requires a mathematical tool called **calculus of variations**, described in section 19.4.2. It is not necessary to understand calculus of variations to understand the content of this chapter. At the moment, it is only necessary to understand that calculus of variations may be used to derive the following two results.

Our first result derived using calculus of variations is that solving the optimization problem

$$f^* = \arg\min_f \quad , \sim p \quad ||\boldsymbol{y} - f(\boldsymbol{x})||^2 \tag{6.14}$$

yields

$$f^*(\boldsymbol{x}) = \quad \sim p \quad (\mid) [\boldsymbol{y}], \tag{6.15}$$

so long as this function lies within the class we optimize over. In other words, if we could train on infinitely many samples from the true data generating distribution, minimizing the mean squared error cost function would give a function that predicts the mean of $\boldsymbol{y}$ for each value of $\boldsymbol{x}$.

Different cost functions give different statistics. A second result derived using calculus of variations is that

$$f^* = \arg\min_f \quad , \sim p \quad ||\boldsymbol{y} - f(\boldsymbol{x})||_1 \tag{6.16}$$

yields a function that predicts the *median* value of $\boldsymbol{y}$ for each $\boldsymbol{x}$, as long as such a function may be described by the family of functions we optimize over. This cost function is commonly called **mean absolute error**.

Unfortunately, mean squared error and mean absolute error often lead to poor results when used with gradient-based optimization. Some output units that saturate produce very small gradients when combined with these cost functions. This is one reason that the cross-entropy cost function is more popular than mean squared error or mean absolute error, even when it is not necessary to estimate an entire distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$.

The choice of cost function is tightly coupled with the choice of output unit. Most of the time, we simply use the cross-entropy between the data distribution and the model distribution. The choice of how to represent the output then determines the form of the cross-entropy function.

Any kind of neural network unit that may be used as an output can also be used as a hidden unit. Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well. We revisit these units with additional detail about their use as hidden units in section 6.3.

Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The role of the output layer is then to provide some additional transformation from the features to complete the task that the network must perform.

6.2.2.1 Linear Units for Gaussian Output Distributions

One simple kind of output unit is based on an affine transformation with no nonlinearity. These are often just called linear units.

Given features $\boldsymbol{h}$, a layer of linear output units produces a vector $\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$.

Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}). \tag{6.17}$$

Maximizing the log-likelihood is then equivalent to minimizing the mean squared error.

The maximum likelihood framework makes it straightforward to learn the covariance of the Gaussian too, or to make the covariance of the Gaussian be a function of the input. However, the covariance must be constrained to be a positive definite matrix for all inputs. It is difficult to satisfy such constraints with a linear output layer, so typically other output units are used to parametrize the covariance. Approaches to modeling the covariance are described shortly, in section 6.2.2.4.

Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms.

6.2.2.2 Sigmoid Units for Bernoulli Output Distributions

Many tasks require predicting the value of a binary variable y. Classification problems with two classes can be cast in this form.

The maximum likelihood approach is to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$.

A Bernoulli distribution is defined by just a single number. The neural net needs to predict only $P(y = 1 \mid \boldsymbol{x})$. For this number to be a valid probability, it must lie in the interval [0, 1].

Satisfying this constraint requires some careful design effort. Suppose we were to use a linear unit and threshold its value to obtain a valid probability:

$$P(y = 1 \mid \boldsymbol{x}) = \max \quad 0, \min \quad 1, \boldsymbol{w}^\top \boldsymbol{h} + b \quad . \tag{6.18}$$

This would indeed define a valid conditional distribution, but we would not be able to train it very effectively with gradient descent. Any time that $\boldsymbol{w}^\top \boldsymbol{h} + b$ strayed outside the unit interval, the gradient of the output of the model with respect to its parameters would be . A gradient of is typically problematic because the learning algorithm no longer has a guide for how to improve the corresponding parameters.

Instead, it is better to use a different approach that ensures there is always a strong gradient whenever the model has the wrong answer. This approach is based on using sigmoid output units combined with maximum likelihood.

A sigmoid output unit is defined by

$$\hat{y} = \sigma \quad \boldsymbol{w}^\top \boldsymbol{h} + b \quad , \tag{6.19}$$

where σ is the logistic sigmoid function described in section 3.10.

We can think of the sigmoid output unit as having two components. First, it uses a linear layer to compute $z = \boldsymbol{w}^\top \boldsymbol{h} + b$. Next, it uses the sigmoid activation function to convert z into a probability.

We omit the dependence on $\boldsymbol{x}$ for the moment to discuss how to define a probability distribution over y using the value z. The sigmoid can be motivated by constructing an unnormalized probability distribution $\tilde{P}(y)$, which does not sum to 1. We can then divide by an appropriate constant to obtain a valid probability distribution. If we begin with the assumption that the unnormalized log probabilities are linear in y and z, we can exponentiate to obtain the unnormalized probabilities. We then normalize to see that this yields a Bernoulli distribution controlled by a sigmoidal transformation of z:

$$\log \tilde{P}(y) = yz, \tag{6.20}$$

$$\tilde{P}(y) = \exp(yz), \tag{6.21}$$

$$P(y) = \frac{\exp(yz)}{{}^{1}_{y\ =0} \exp(y'z),} \tag{6.22}$$

$$P(y) = \sigma\left((2y-1)z\right). \tag{6.23}$$

Probability distributions based on exponentiation and normalization are common throughout the statistical modeling literature. The z variable defining such a distribution over binary variables is called a **logit**.

This approach to predicting the probabilities in log space is natural to use with maximum likelihood learning. Because the cost function used with maximum likelihood is $-\log P(y \mid \boldsymbol{x})$, the log in the cost function undoes the exp of the sigmoid. Without this effect, the saturation of the sigmoid could prevent gradient-based learning from making good progress. The loss function for maximum likelihood learning of a Bernoulli parametrized by a sigmoid is

$$J(\boldsymbol{\theta}) = -\log P(y \mid \boldsymbol{x}) \tag{6.24}$$
$$= -\log \sigma\left((2y-1)z\right) \tag{6.25}$$
$$= \zeta\left((1-2y)z\right). \tag{6.26}$$

This derivation makes use of some properties from section 3.10. By rewriting the loss in terms of the softplus function, we can see that it saturates only when $(1-2y)z$ is very negative. Saturation thus occurs only when the model already has the right answer—when $y = 1$ and z is very positive, or $y = 0$ and z is very negative. When z has the wrong sign, the argument to the softplus function, $(1-2y)z$, may be simplified to $|z|$. As $|z|$ becomes large while z has the wrong sign, the softplus function asymptotes toward simply returning its argument $|z|$. The derivative with respect to z asymptotes to $\text{sign}(z)$, so, in the limit of extremely incorrect z, the softplus function does not shrink the gradient at all. This property is useful because it means that gradient-based learning can act to quickly correct a mistaken z.

When we use other loss functions, such as mean squared error, the loss can saturate anytime $\sigma(z)$ saturates. The sigmoid activation function saturates to 0 when z becomes very negative and saturates to 1 when z becomes very positive. The gradient can shrink too small to be useful for learning when this happens, whether the model has the correct answer or the incorrect answer. For this reason, maximum likelihood is almost always the preferred approach to training sigmoid output units.

Analytically, the logarithm of the sigmoid is always defined and finite, because the sigmoid returns values restricted to the open interval $(0, 1)$, rather than using the entire closed interval of valid probabilities $[0, 1]$. In software implementations, to avoid numerical problems, it is best to write the negative log-likelihood as a function of z, rather than as a function of $\hat{y} = \sigma(z)$. If the sigmoid function underflows to zero, then taking the logarithm of $\hat{y}$ yields negative infinity.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. This can be seen as a generalization of the sigmoid function, which was used to represent a probability distribution over a binary variable.

Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.

In the case of binary variables, we wished to produce a single number

$$\hat{y} = P(y = 1 \mid \boldsymbol{x}). \tag{6.27}$$

Because this number needed to lie between 0 and 1, and because we wanted the logarithm of the number to be well behaved for gradient-based optimization of the log-likelihood, we chose to instead predict a number $z = \log \tilde{P}(y = 1 \mid \boldsymbol{x})$. Exponentiating and normalizing gave us a Bernoulli distribution controlled by the sigmoid function.

To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. We require not only that each element of $\hat{y}_i$ be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution. The same approach that worked for the Bernoulli distribution generalizes to the multinoulli distribution. First, a linear layer predicts unnormalized log probabilities:

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}, \tag{6.28}$$

where $z_i = \log \tilde{P}(y = i \mid \boldsymbol{x})$. The softmax function can then exponentiate and normalize $\boldsymbol{z}$ to obtain the desired $\hat{\boldsymbol{y}}$. Formally, the softmax function is given by

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{{}_j \exp(z_j)}. \tag{6.29}$$

As with the logistic sigmoid, the use of the exp function works well when training the softmax to output a target value y using maximum log-likelihood. In this case, we wish to maximize $\log P(\mathrm{y} = i; \boldsymbol{z}) = \log \text{softmax}(\boldsymbol{z})_i$. Defining the softmax in terms of exp is natural because the log in the log-likelihood can undo the exp of the softmax:

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \underset{j}{} \exp(z_j). \tag{6.30}$$

The first term of equation 6.30 shows that the input z_i always has a direct contribution to the cost function. Because this term cannot saturate, we know that learning can proceed, even if the contribution of z_i to the second term of equation 6.30 becomes very small. When maximizing the log-likelihood, the first term encourages z_i to be pushed up, while the second term encourages all of $\boldsymbol{z}$ to be pushed down. To gain some intuition for the second term, $\log \;\; {}_j \exp(z_j)$, observe that this term can be roughly approximated by $\max_j z_j$. This approximation is based on the idea that $\exp(z_k)$ is insignificant for any z_k that is noticeably less than $\max_j z_j$. The intuition we can gain from this approximation is that the negative log-likelihood cost function always strongly penalizes the most active incorrect prediction. If the correct answer already has the largest input to the softmax, then the $-z_i$ term and the $\log \; {}_j \exp(z_j) \approx \max_j z_j = \; z_i$ terms will roughly cancel. This example will then contribute little to the overall training cost, which will be dominated by other examples that are not yet correctly classified.

So far we have discussed only a single example. Overall, unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict the fraction of counts of each outcome observed in the training set:

$$\text{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))_i \approx \frac{{}^{\;\;m}_{j=1}\;\; y \;\; =i, \qquad =}{{}^{\;\;m}_{j=1} \qquad\quad =}. \tag{6.31}$$

Because maximum likelihood is a consistent estimator, this is guaranteed to happen as long as the model family is capable of representing the training distribution. In practice, limited model capacity and imperfect optimization will mean that the model is only able to approximate these fractions.

Many objective functions other than the log-likelihood do not work as well with the softmax function. Specifically, objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative, causing the gradient to vanish. In particular, squared error is a poor loss function for softmax units and can fail to train the model to change its output, even when the model makes highly confident incorrect predictions (Bridle, 1990). To understand why these other loss functions can fail, we need to examine the softmax function itself.

Like the sigmoid, the softmax activation can saturate. The sigmoid function has a single output that saturates when its input is extremely negative or extremely positive. The softmax has multiple output values. These output values can saturate when the differences between input values become extreme. When the softmax saturates, many cost functions based on the softmax also saturate, unless they are able to invert the saturating activating function.

To see that the softmax function responds to the difference between its inputs, observe that the softmax output is invariant to adding the same scalar to all its inputs:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} + c). \tag{6.32}$$

Using this property, we can derive a numerically stable variant of the softmax:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} - \max_i z_i). \tag{6.33}$$

The reformulated version enables us to evaluate softmax with only small numerical errors, even when z contains extremely large or extremely negative numbers. Examining the numerically stable variant, we see that the softmax function is driven by the amount that its arguments deviate from $\max_i z_i$.

An output $\text{softmax}(\boldsymbol{z})_i$ saturates to 1 when the corresponding input is maximal ($z_i = \max_i z_i$) and z_i is much greater than all the other inputs. The output $\text{softmax}(\boldsymbol{z})_i$ can also saturate to 0 when z_i is not maximal and the maximum is much greater. This is a generalization of the way that sigmoid units saturate and can cause similar difficulties for learning if the loss function is not designed to compensate for it.

The argument $\boldsymbol{z}$ to the softmax function can be produced in two different ways. The most common is simply to have an earlier layer of the neural network output every element of $\boldsymbol{z}$, as described above using the linear layer $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$. While straightforward, this approach actually overparametrizes the distribution. The constraint that the n outputs must sum to 1 means that only $n - 1$ parameters are necessary; the probability of the n-th value may be obtained by subtracting the first $n - 1$ probabilities from 1. We can thus impose a requirement that one element of $\boldsymbol{z}$ be fixed. For example, we can require that $z_n = 0$. Indeed, this is exactly what the sigmoid unit does. Defining $P(y = 1 \mid \boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1 \mid \boldsymbol{x}) = \text{softmax}(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and $z_1 = 0$. Both the $n - 1$ argument and the n argument approaches to the softmax can describe the same set of probability distributions but have different learning dynamics. In practice, there is rarely much difference between using the overparametrized version or the restricted version, and it is simpler to implement the overparametrized version.

From a neuroscientific point of view, it is interesting to think of the softmax as a way to create a form of competition between the units that participate in it: the softmax outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decrease in the value of others. This is analogous to the lateral inhibition that is believed to exist between nearby neurons in the cortex. At the extreme (when the difference between the maximal a_i and the others is large in magnitude) it becomes a form of **winner-take-all** (one of the outputs is nearly 1, and the others are nearly 0).

The name "softmax" can be somewhat confusing. The function is more closely related to the $\arg\max$ function than the max function. The term "soft" derives from the fact that the softmax function is continuous and differentiable. The $\arg\max$ function, with its result represented as a one-hot vector, is not continuous or differentiable. The softmax function thus provides a "softened" version of the $\arg\max$. The corresponding soft version of the maximum function is $\text{softmax}(\boldsymbol{z})^\top \boldsymbol{z}$.

It would perhaps be better to call the softmax function "softargmax," but the current name is an entrenched convention.

6.2.2.4 Other Output Types

The linear, sigmoid, and softmax output units described above are the most common. Neural networks can generalize to almost any kind of output layer that we wish. The principle of maximum likelihood provides a guide for how to design a good cost function for nearly any kind of output layer.

In general, if we define a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, the principle of maximum likelihood suggests we use $-\log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ as our cost function.

In general, we can think of the neural network as representing a function $f(\boldsymbol{x}; \boldsymbol{\theta})$. The outputs of this function are not direct predictions of the value $\boldsymbol{y}$. Instead, $f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$ provides the parameters for a distribution over y. Our loss function can then be interpreted as $-\log p(\ ; \boldsymbol{\omega}(\boldsymbol{x}))$.

For example, we may wish to learn the variance of a conditional Gaussian for , given . In the simple case, where the variance σ^2 is a constant, there is a closed form expression because the maximum likelihood estimator of variance is simply the empirical mean of the squared difference between observations and their expected value. A computationally more expensive approach that does not require writing special-case code is to simply include the variance as one of the properties of the distribution $p(\ \mid \boldsymbol{x})$ that is controlled by $\boldsymbol{\omega} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The negative log-likelihood $-\log p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$ will then provide a cost function with the appropriate terms necessary to make our optimization procedure incrementally learn the variance. In the simple case where the standard deviation does not depend on the input, we can make a new parameter in the network that is copied directly into $\boldsymbol{\omega}$. This new parameter might be σ itself or could be a parameter v representing σ^2 or it could be a parameter β representing $\frac{1}{\sigma}$, depending on how we choose to parametrize the distribution. We may wish our model to predict a different amount of variance in for different values of . This is called a **heteroscedastic** model. In the heteroscedastic case, we simply make the specification of the variance be one of the values output by $f(\ ; \boldsymbol{\theta})$. A typical way to do this is to formulate the Gaussian distribution using precision, rather than variance, as described in equation 3.22. In the multivariate case, it is most common to use a diagonal precision matrix

$$\operatorname{diag}(\boldsymbol{\beta}). \tag{6.34}$$

This formulation works well with gradient descent because the formula for the log-likelihood of the Gaussian distribution parametrized by $\boldsymbol{\beta}$ involves only multiplication by β_i and addition of $\log \boldsymbol{\beta}_i$. The gradient of multiplication, addition, and logarithm operations is well behaved. By comparison, if we parametrized the output in terms of variance, we would need to use division. The division function becomes arbitrarily steep near zero. While large gradients can help learning, arbitrarily large gradients usually result in instability. If we parametrized the output in terms of standard deviation, the log-likelihood would still involve division as well as squaring. The gradient through the squaring operation can vanish near zero, making it difficult to learn parameters that are squared. Regardless of whether we use standard deviation, variance, or precision, we must ensure that the covariance matrix of the Gaussian is positive definite. Because the eigenvalues of the precision matrix are the reciprocals of the eigenvalues of the covariance matrix, this is equivalent to ensuring that the precision matrix is positive definite. If we use a diagonal matrix, or a scalar times the diagonal matrix, then the only condition we need to enforce on the output of the model is positivity. If we suppose that $\boldsymbol{a}$ is the raw activation of the model used to determine the diagonal precision, we can use the softplus function to obtain a positive precision vector: $\boldsymbol{\beta} = \zeta(\boldsymbol{a})$. This same strategy applies equally if using variance or standard deviation rather than precision or if using a scalar times identity rather than diagonal matrix.

It is rare to learn a covariance or precision matrix with richer structure than diagonal. If the covariance is full and conditional, then a parametrization must be chosen that guarantees positive definiteness of the predicted covariance matrix. This can be achieved by writing $\;(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^{\top}(\boldsymbol{x})$, where $\boldsymbol{B}$ is an unconstrained square matrix. One practical issue if the matrix is full rank is that computing the likelihood is expensive, with a $d \times d$ matrix requiring $O(d^3)$ computation for the determinant and inverse of $\;(\boldsymbol{x})$ (or equivalently, and more commonly done, its eigendecomposition or that of $\boldsymbol{B}(\boldsymbol{x})$).

We often want to perform multimodal regression, that is, to predict real values from a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ that can have several different peaks in $\boldsymbol{y}$ space for the same value of $\boldsymbol{x}$. In this case, a Gaussian mixture is a natural representation for the output (Jacobs *et al.*, 1991; Bishop, 1994). Neural networks with Gaussian mixtures as their output are often called **mixture density networks**. A Gaussian mixture output with n components is defined by the conditional probability distribution:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathop{}_{i=1}^{n} p(\mathrm{c} = i \mid \boldsymbol{x})\mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}^{(i)}(\boldsymbol{x}), \;^{(i)}(\boldsymbol{x})). \tag{6.35}$$

The neural network must have three outputs: a vector defining $p(\mathrm{c} = i \mid \boldsymbol{x})$, a matrix providing $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$ for all i, and a tensor providing $\;^{(i)}(\boldsymbol{x})$ for all i. These outputs must satisfy different constraints:

1. Mixture components $p(\mathrm{c} = i \mid \boldsymbol{x})$: these form a multinoulli distribution over the n different components associated with latent variable c, and can typically be obtained by a softmax over an n-dimensional vector, to guarantee that these outputs are positive and sum to 1.

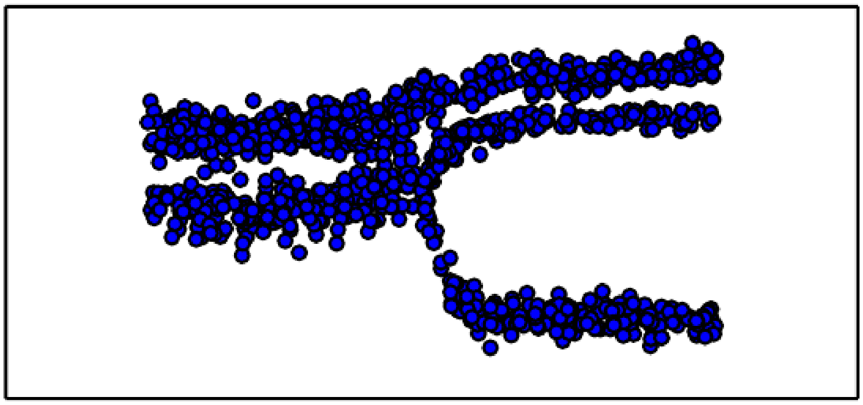
2. Means $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$: these indicate the center or mean associated with the i-th Gaussian component and are unconstrained (typically with no nonlinearity at all for these output units). If is a d-vector, then the network must output an $n \times d$ matrix containing all n of these d-dimensional vectors. Learning these means with maximum likelihood is slightly more complicated than learning the means of a distribution with only one output mode. We only want to update the mean for the component that actually produced the observation. In practice, we do not know which component produced each observation. The expression for the negative log-likelihood naturally weights each example's contribution to the loss for each component by the probability that the component produced the example.

3. Covariances $^{(i)}(\boldsymbol{x})$: these specify the covariance matrix for each component i. As when learning a single Gaussian component, we typically use a diagonal matrix to avoid needing to compute determinants. As with learning the means of the mixture, maximum likelihood is complicated by needing to assign partial responsibility for each point to each mixture component. Gradient descent will automatically follow the correct process if given the correct specification of the negative log-likelihood under the mixture model.

It has been reported that gradient-based optimization of conditional Gaussian mixtures (on the output of neural networks) can be unreliable, in part because one gets divisions (by the variance) which can be numerically unstable (when some variance gets to be small for a particular example, yielding very large gradients). One solution is to **clip gradients** (see section 10.11.1), while another is to scale the gradients heuristically (Uria *et al.*, 2014).

Gaussian mixture outputs are particularly effective in generative models of speech (Schuster, 1999) and movements of physical objects (Graves, 2013). The mixture density strategy gives a way for the network to represent multiple output modes and to control the variance of its output, which is crucial for obtaining

We consider c to be latent because we do not observe it in the data: given input and target , it is not possible to know with certainty which Gaussian component was responsible for , but we can imagine that was generated by picking one of them, and we can make that unobserved choice a random variable.



x y
p $(y \mid x)$

x

a high degree of quality in these real-valued domains. An example of a mixture density network is shown in figure 6.4.

In general, we may wish to continue to model larger vectors $\boldsymbol{y}$ containing more variables, and to impose richer and richer structures on these output variables. For example, if we want our neural network to output a sequence of characters that forms a sentence, we might continue to use the principle of maximum likelihood applied to our model $p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$. In this case, the model we use to describe $\boldsymbol{y}$ would become complex enough to be beyond the scope of this chapter. In Chapter 10 we describe how to use recurrent neural networks to define such models over sequences, and in part III we describe advanced techniques for modeling arbitrary probability distributions.

So far we have focused our discussion on design choices for neural networks that are common to most parametric machine learning models trained with gradient-based optimization. Now we turn to an issue that is unique to feedforward neural networks: how to choose the type of hidden unit to use in the hidden layers of the model.

The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles.

Rectified linear units are an excellent default choice of hidden unit. Many other types of hidden units are available. It can be difficult to determine when to use which kind (though rectified linear units are usually an acceptable choice). We describe here some of the basic intuitions motivating each type of hidden unit. These intuitions can help decide when to try out which unit. Predicting in advance which will work best is usually impossible. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

Some of the hidden units included in this list are not actually differentiable at all input points. For example, the rectified linear function $g(z) = \max\{0, z\}$ is not differentiable at $z = 0$. This may seem like it invalidates g for use with a gradient-based learning algorithm. In practice, gradient descent still performs well enough for these models to be used for machine learning tasks. This is in part because neural network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly, as shown in figure 4.3. (These ideas are described further in chapter 8.) Because we do not expect training to actually reach a point where the gradient is , it is acceptable for the minima of the cost function to correspond to points with undefined gradient. Hidden units that are not differentiable are usually nondifferentiable at only a small number of points. In general, a function $g(z)$ has a left derivative defined by the slope of the function immediately to the left of z and a right derivative defined by the slope of the function immediately to the right of z. A function is differentiable at z only if both the left derivative and the right derivative are defined and equal to each other. The functions used in the context of neural networks usually have defined left derivatives and defined right derivatives. In the case of $g(z) = \max\{0, z\}$, the left derivative at $z = 0$ is 0, and the right derivative is 1. Software implementations of neural network training usually return one of the one-sided derivatives rather than reporting that the derivative is undefined or raising an error. This may be heuristically justified by observing that gradient-based optimization on a digital computer is subject to numerical error anyway. When a function is asked to evaluate $g(0)$, it is very unlikely that the underlying value truly was 0. Instead, it was likely to be some small value ϵ that was rounded to 0. In some contexts, more theoretically pleasing justifications are available, but these usually do not apply to neural network training. The important point is that in practice one can safely disregard the nondifferentiability of the hidden unit activation functions described below.

Unless indicated otherwise, most hidden units can be described as accepting a vector of inputs $\boldsymbol{x}$, computing an affine transformation $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$, and then applying an element-wise nonlinear function $g(\boldsymbol{z})$. Most hidden units are distinguished from each other only by the choice of the form of the activation function $g(\boldsymbol{z})$.

Rectified linear units use the activation function $g(z) = \max\{0, z\}$.

These units are easy to optimize because they are so similar to linear units. The only difference between a linear unit and a rectified linear unit is that a rectified linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active. The gradients are not only large but also consistent. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active. This means that the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects.

Rectified linear units are typically used on top of an affine transformation:

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}). \tag{6.36}$$

When initializing the parameters of the affine transformation, it can be a good practice to set all elements of $\boldsymbol{b}$ to a small positive value, such as 0.1. Doing so makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.

Several generalizations of rectified linear units exist. Most of these generalizations perform comparably to rectified linear units and occasionally perform better.

One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero. Various generalizations of rectified linear units guarantee that they receive gradient everywhere.

Three generalizations of rectified linear units are based on using a nonzero slope α_i when $z_i < 0$: $h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$. **Absolute value**

rectification fixes $\alpha_i = -1$ to obtain $g(z) = |z|$. It is used for object recognition from images (Jarrett *et al.*, 2009), where it makes sense to seek features that are invariant under a polarity reversal of the input illumination. Other generalizations of rectified linear units are more broadly applicable. A **leaky ReLU** (Maas *et al.*, 2013) fixes α_i to a small value like 0.01, while a **parametric ReLU**, or **PReLU**, treats α_i as a learnable parameter (He *et al.*, 2015).

Maxout units (Goodfellow *et al.*, 2013a) generalize rectified linear units further. Instead of applying an element-wise function $g(z)$, maxout units divide $\boldsymbol{z}$ into groups of k values. Each maxout unit then outputs the maximum element of one of these groups:

$$g(\boldsymbol{z})_i = \max_{j \in} z_j, \tag{6.37}$$

where $^{(i)}$ is the set of indices into the inputs for group i, $\{(i-1)k+1, \ldots, ik\}$. This provides a way of learning a piecewise linear function that responds to multiple directions in the input $\boldsymbol{x}$ space.

A maxout unit can learn a piecewise linear, convex function with up to k pieces. Maxout units can thus be seen as *learning the activation function* itself rather than just the relationship between units. With large enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity. In particular, a maxout layer with two pieces can learn to implement the same function of the input $\boldsymbol{x}$ as a traditional layer using the rectified linear activation function, the absolute value rectification function, or the leaky or parametric ReLU, or it can learn to implement a totally different function altogether. The maxout layer will of course be parametrized differently from any of these other layer types, so the learning dynamics will be different even in the cases where maxout learns to implement the same function of $\boldsymbol{x}$ as one of the other layer types.

Each maxout unit is now parametrized by k weight vectors instead of just one, so maxout units typically need more regularization than rectified linear units. They can work well without regularization if the training set is large and the number of pieces per unit is kept low (Cai *et al.*, 2013).

Maxout units have a few other benefits. In some cases, one can gain some statistical and computational advantages by requiring fewer parameters. Specifically, if the features captured by n different linear filters can be summarized without losing information by taking the max over each group of k features, then the next layer can get by with k times fewer weights.

Because each unit is driven by multiple filters, maxout units have some redundancy that helps them resist a phenomenon called **catastrophic forgetting**, in which neural networks forget how to perform tasks that they were trained on in the past (Goodfellow *et al.*, 2014a).

Rectified linear units and all these generalizations of them are based on the principle that models are easier to optimize if their behavior is closer to linear. This same general principle of using linear behavior to obtain easier optimization also applies in other contexts besides deep linear networks. Recurrent networks can learn from sequences and produce a sequence of states and outputs. When training them, one needs to propagate information through several time steps, which is much easier when some linear computations (with some directional derivatives being of magnitude near 1) are involved. One of the best-performing recurrent network architectures, the LSTM, propagates information through time via summation—a particular straightforward kind of linear activation. This is discussed further in section 10.10.

Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function

$$g(z) = \sigma(z) \tag{6.38}$$

or the hyperbolic tangent activation function

$$g(z) = \tanh(z). \tag{6.39}$$

These activation functions are closely related because $\tanh(z) = 2\sigma(2z) - 1$.

We have already seen sigmoid units as output units, used to predict the probability that a binary variable is 1. Unlike piecewise linear units, sigmoidal units saturate across most of their domain—they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are only strongly sensitive to their input when z is near 0. The widespread saturation of sigmoidal units can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged. Their use as output units is compatible with the use of gradient-based learning when an appropriate cost function can undo the saturation of the sigmoid in the output layer.

When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic sigmoid. It resembles the identity function more closely, in the sense that $\tanh(0) = 0$ while $\sigma(0) = \frac{1}{2}$. Because tanh is similar to the identity function near 0, training a deep neural network $\hat{y} = \boldsymbol{w}^\top \tanh(\boldsymbol{U}^\top \tanh(\boldsymbol{V}^\top \boldsymbol{x}))$ resembles training a linear model $\hat{y} =$

$\boldsymbol{w}^\top \boldsymbol{U}^\top \boldsymbol{V}^\top \boldsymbol{x}$ as long as the activations of the network can be kept small. This makes training the tanh network easier.

Sigmoidal activation functions are more common in settings other than feed-forward networks. Recurrent networks, many probabilistic models, and some autoencoders have additional requirements that rule out the use of piecewise linear activation functions and make sigmoidal units more appealing despite the drawbacks of saturation.

Many other types of hidden units are possible but are used less frequently.

In general, a wide variety of differentiable functions perform perfectly well. Many unpublished activation functions perform just as well as the popular ones. To provide a concrete example, we tested a feedforward network using $\boldsymbol{h} = \cos(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ on the MNIST dataset and obtained an error rate of less than 1 percent, which is competitive with results obtained using more conventional activation functions. During research and development of new techniques, it is common to test many different activation functions and find that several variations on standard practice perform comparably. This means that usually new hidden unit types are published only if they are clearly demonstrated to provide a significant improvement. New hidden unit types that perform roughly comparably to known types are so common as to be uninteresting.

It would be impractical to list all the hidden unit types that have appeared in the literature. We highlight a few especially useful and distinctive ones.

One possibility is to not have an activation $g(z)$ at all. One can also think of this as using the identity function as the activation function. We have already seen that a linear unit can be useful as the output of a neural network. It may also be used as a hidden unit. If every layer of the neural network consists of only linear transformations, then the network as a whole will be linear. However, it is acceptable for some layers of the neural network to be purely linear. Consider a neural network layer with n inputs and p outputs, $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$. We may replace this with two layers, with one layer using weight matrix $\boldsymbol{U}$ and the other using weight matrix $\boldsymbol{V}$. If the first layer has no activation function, then we have essentially factored the weight matrix of the original layer based on $\boldsymbol{W}$. The factored approach is to compute $\boldsymbol{h} = g(\boldsymbol{V}^\top \boldsymbol{U}^\top \boldsymbol{x} + \boldsymbol{b})$. If $\boldsymbol{U}$ produces q outputs, then $\boldsymbol{U}$ and $\boldsymbol{V}$ together contain only $(n + p)q$ parameters, while $\boldsymbol{W}$ contains np parameters. For small q, this can be a considerable saving in parameters. It comes at the cost of constraining the linear transformation to be low rank, but these low-rank relationships are often sufficient. Linear hidden units thus offer an effective way of reducing the number of parameters in a network.

Softmax units are another kind of unit that is usually used as an output (as described in section 6.2.2.3) but may sometimes be used as a hidden unit. Softmax units naturally represent a probability distribution over a discrete variable with k possible values, so they may be used as a kind of switch. These kinds of hidden units are usually only used in more advanced architectures that explicitly learn to manipulate memory, as described in section 10.12.

A few other reasonably common hidden unit types include

- **Radial basis function** (RBF), unit: $h_i = \exp \; -\frac{1}{\sigma}||\boldsymbol{W}_{:,i} - \boldsymbol{x}||^2$. This function becomes more active as $\boldsymbol{x}$ approaches a template $\boldsymbol{W}_{:,i}$. Because it saturates to 0 for most $\boldsymbol{x}$, it can be difficult to optimize.

- **Softplus**: $g(a) = \zeta(a) = \log(1+e^a)$. This is a smooth version of the rectifier, introduced by Dugas *et al.* (2001) for function approximation and by Nair and Hinton (2010) for the conditional distributions of undirected probabilistic models. Glorot *et al.* (2011a) compared the softplus and rectifier and found better results with the latter. The use of the softplus is generally discouraged. The softplus demonstrates that the performance of hidden unit types can be very counterintuitive—one might expect it to have an advantage over the rectifier due to being differentiable everywhere or due to saturating less completely, but empirically it does not.

- **Hard tanh**. This is shaped similarly to the tanh and the rectifier, but unlike the latter, it is bounded, $g(a) = \max(-1, \min(1, a))$. It was introduced by Collobert (2004).

Hidden unit design remains an active area of research, and many useful hidden unit types remain to be discovered.

Another key design consideration for neural networks is determining the architecture. The word **architecture** refers to the overall structure of the network: how many units it should have and how these units should be connected to each other.

Most neural networks are organized into groups of units called layers. Most neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In this structure, the first layer is given by

$$\boldsymbol{h}^{(1)} = g^{(1)} \quad \boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)} \quad ; \tag{6.40}$$

the second layer is given by

$$\boldsymbol{h}^{(2)} = g^{(2)} \quad \boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)} \quad ; \tag{6.41}$$

and so on.

In these chain-based architectures, the main architectural considerations are choosing the depth of the network and the width of each layer. As we will see, a network with even one hidden layer is sufficient to fit the training set. Deeper networks are often able to use far fewer units per layer and far fewer parameters, as well as frequently generalizing to the test set, but they also tend to be harder to optimize. The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error.

A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. It has the advantage of being easy to train because many loss functions result in convex optimization problems when applied to linear models. Unfortunately, we often want our systems to learn nonlinear functions.

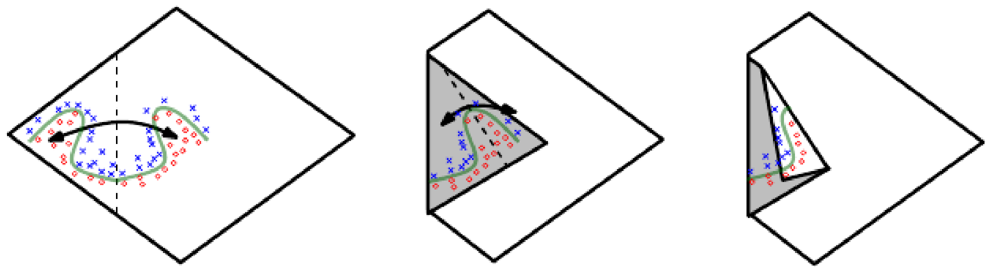
At first glance, we might presume that learning a nonlinear function requires designing a specialized model family for the kind of nonlinearity we want to learn. Fortunately, feedforward networks with hidden layers provide a universal approximation framework. Specifically, the **universal approximation theorem** (Hornik *et al.*, 1989; Cybenko, 1989) states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (Hornik *et al.*, 1990). The concept of Borel measurability is beyond the scope of this book; for our purposes it suffices to say that any continuous function on a closed and bounded subset of ^{n} is Borel measurable and therefore may be approximated by a neural network. A neural network may also approximate any function mapping from any finite dimensional discrete space to another. While the original theorems were first stated in terms of units with activation functions that saturate for both very negative and very positive arguments, universal approximation theorems have also been proved for a wider class of activation functions, which includes the now commonly used rectified linear unit (Leshno *et al.*, 1993).

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to *represent* this function. We are not guaranteed, however, that the training algorithm will be able to *learn* that function. Even if the MLP is able to represent the function, learning can fail for two different reasons. First, the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm might choose the wrong function as a result of overfitting. Recall from section 5.2.1 that the no free lunch theorem shows that there is no universally superior machine learning algorithm. Feedforward networks provide a universal system for representing functions in the sense that, given a function, there exists a feedforward network that approximates the function. There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

According to the universal approximation theorem, there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be. Barron (1993) provides some bounds on the size of a single-layer network needed to approximate a broad class of functions. Unfortunately, in the worst case, an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs to be distinguished) may be required. This is easiest to see in the binary case: the number of possible binary functions on vectors $\boldsymbol{v} \in \{0, 1\}^n$ is 2^2 and selecting one such function requires 2^n bits, which will in general require $O(2^n)$ degrees of freedom.

In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.

Various families of functions can be approximated efficiently by an architecture with depth greater than some value d, but they require a much larger model if depth is restricted to be less than or equal to d. In many cases, the number of hidden units required by the shallow model is exponential in n. Such results



were first proved for models that do not resemble the continuous, differentiable neural networks used for machine learning but have since been extended to these models. The first results were for circuits of logic gates (Håstad, 1986). Later work extended these results to linear threshold units with nonnegative weights (Håstad and Goldmann, 1991; Hajnal *et al.*, 1993), and then to networks with continuous-valued activations (Maass, 1992; Maass *et al.*, 1994). Many modern neural networks use rectified linear units. Leshno *et al.* (1993) demonstrated that shallow networks with a broad family of non-polynomial activation functions, including rectified linear units, have universal approximation properties, but these results do not address the questions of depth or efficiency—they specify only that a sufficiently wide rectifier network could represent any function. Montufar *et al.* (2014) showed that functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network. More precisely, they showed that piecewise linear networks (which can be obtained from rectifier nonlinearities or maxout units) can represent functions with a number of regions that is exponential in the depth of the network. Figure 6.5 illustrates how a network with absolute value rectification creates mirror images of the function computed on top of some hidden unit, with respect to the input of that hidden unit. Each hidden unit specifies where to fold the input space in order to create mirror responses (on both sides of the absolute value nonlinearity). By composing these folding operations, we obtain an exponentially large number of piecewise linear regions that can capture all kinds of regular (e.g., repeating) patterns.

The main theorem in Montufar *et al.* (2014) states that the number of linear regions carved out by a deep rectifier network with d inputs, depth l, and n units per hidden layer is

$$O \quad \frac{n}{d}^{\,d(l-1)} \quad n^d \quad , \tag{6.42}$$

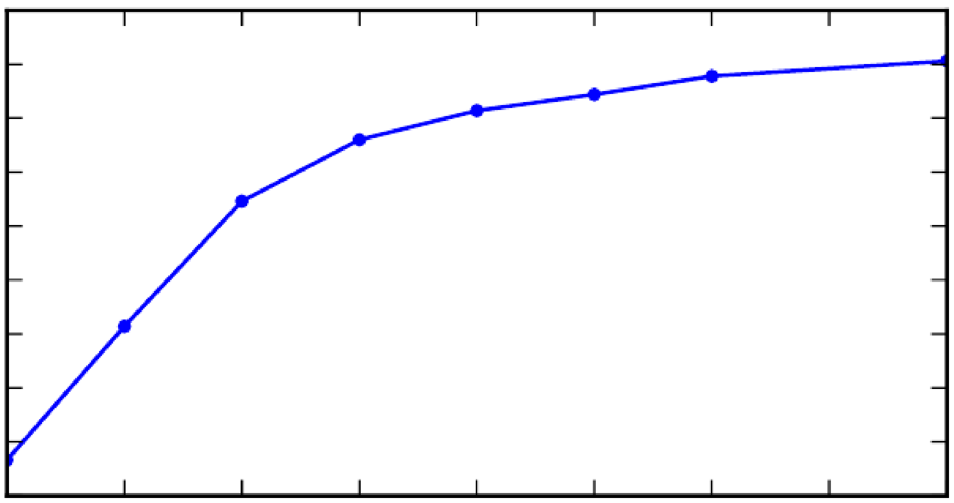
that is, exponential in depth l. In the case of maxout networks with k filters per unit, the number of linear regions is

$$O \quad k^{(l-1)+d} \quad . \tag{6.43}$$

Of course, there is no guarantee that the kinds of functions we want to learn in applications of machine learning (and in particular for AI) share such a property.

We may also want to choose a deep model for statistical reasons. Any time we choose a specific machine learning algorithm, we are implicitly stating some set of prior beliefs we have about what kind of function the algorithm should learn. Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation. Alternately, we can interpret the use of a deep architecture as expressing a belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous step's output. These intermediate outputs are not necessarily factors of variation but can instead be analogous to counters or pointers that the network uses to organize its internal processing. Empirically, greater depth does seem to result in better generalization for a wide variety of tasks (Bengio *et al.*, 2007; Erhan *et al.*, 2009; Bengio, 2009; Mesnil *et al.*, 2011; Ciresan *et al.*, 2012; Krizhevsky *et al.*, 2012; Sermanet *et al.*, 2013; Farabet *et al.*, 2013; Couprie *et al.*, 2013; Kahou *et al.*, 2013; Goodfellow *et al.*, 2014d; Szegedy *et al.*, 2014a). See figure 6.6 and figure 6.7 for examples of some of these empirical results. These results suggest that using deep architectures does indeed express a useful prior over the space of functions the model learns.

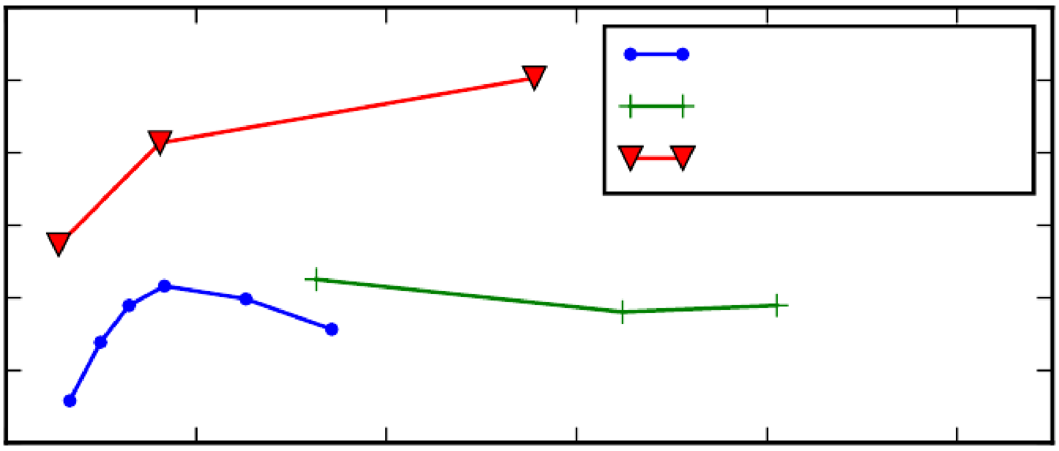
So far we have described neural networks as being simple chains of layers, with the main considerations being the depth of the network and the width of each layer. In practice, neural networks show considerably more diversity.



Many neural network architectures have been developed for specific tasks. Specialized architectures for computer vision called convolutional networks are described in chapter 9. Feedforward networks may also be generalized to the recurrent neural networks for sequence processing, described in chapter 10, which have their own architectural considerations.

In general, the layers need not be connected in a chain, even though this is the most common practice. Many architectures build a main chain but then add extra architectural features to it, such as skip connections going from layer i to layer $i+2$ or higher. These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.

Another key consideration of architecture design is exactly how to connect a pair of layers to each other. In the default neural network layer described by a linear transformation via a matrix $\boldsymbol{W}$, every input unit is connected to every output unit. Many specialized networks in the chapters ahead have fewer connections, so that each unit in the input layer is connected to only a small subset of units in the output layer. These strategies for decreasing the number of connections reduce the number of parameters and the amount of computation required to evaluate the network but are often highly problem dependent. For example, convolutional



networks, described in chapter 9, use specialized patterns of sparse connections that are very effective for computer vision problems. In this chapter, it is difficult to give more specific advice concerning the architecture of a generic neural network. In subsequent chapters we develop the particular architectural strategies that have been found to work well for different application domains.

When we use a feedforward neural network to accept an input $\boldsymbol{x}$ and produce an output $\hat{\boldsymbol{y}}$, information flows forward through the network. The input $\boldsymbol{x}$ provides the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\boldsymbol{y}}$. This is called **forward propagation**. During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$. The **back-propagation** algorithm (Rumelhart *et al.*, 1986a), often simply called **backprop**, allows the information from the cost to then flow backward through the network in order to compute the gradient.

Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.

The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. Furthermore, back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function (for some functions, the correct response is to report that the derivative of the function is undefined). Specifically, we will describe how to compute the gradient $\nabla \ f(\boldsymbol{x}, \boldsymbol{y})$ for an arbitrary function f, where $\boldsymbol{x}$ is a set of variables whose derivatives are desired, and $\boldsymbol{y}$ is an additional set of variables that are inputs to the function but whose derivatives are not required. In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla \ J(\boldsymbol{\theta})$. Many machine learning tasks involve computing other derivatives, either as part of the learning process, or to analyze the learned model. The back-propagation algorithm can be applied to these tasks as well and is not restricted to computing the gradient of the cost function with respect to the parameters. The idea of computing derivatives by propagating information through a network is very general and can be used to compute values such as the Jacobian of a function

f with multiple outputs. We restrict our description here to the most commonly used case, where f has a single output.

So far we have discussed neural networks with a relatively informal graph language. To describe the back-propagation algorithm more precisely, it is helpful to have a more precise **computational graph** language.

Many ways of formalizing computation as graphs are possible.

Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.

To formalize our graphs, we also need to introduce the idea of an **operation**. An operation is a simple function of one or more variables. Our graph language is accompanied by a set of allowable operations. Functions more complicated than the operations in this set may be described by composing many operations together.

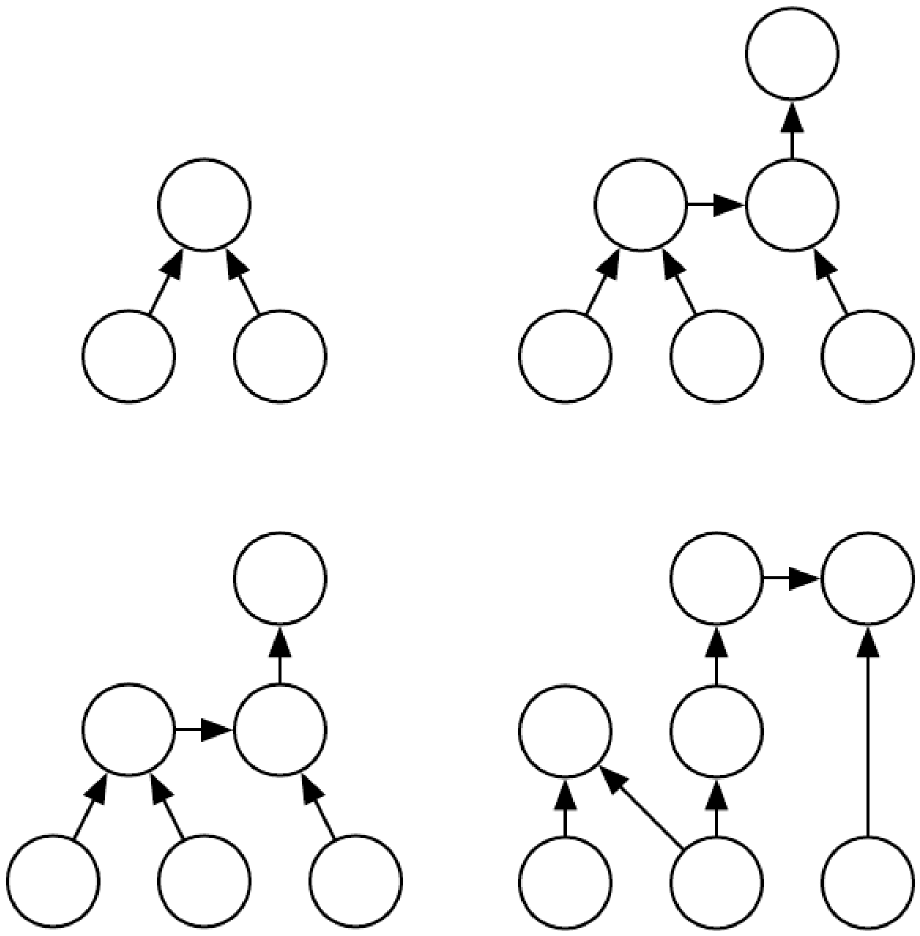
Without loss of generality, we define an operation to return only a single output variable. This does not lose generality because the output variable can have multiple entries, such as a vector. Software implementations of back-propagation usually support operations with multiple outputs, but we avoid this case in our description because it introduces many extra details that are not important to conceptual understanding.

If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y. We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

Examples of computational graphs are shown in figure 6.8.

The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then



$\times$

$z = xy$ $\hat{y} = \sigma \quad \boldsymbol{x} \quad \boldsymbol{w} + b$

i $\boldsymbol{u}$

$\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$

$\boldsymbol{H}$

$\boldsymbol{X}$

$\boldsymbol{w}$

$\hat{y}$ λ w

the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \in {}^m$, $\boldsymbol{y} \in {}^n$, g maps from ^{m} to ^{n}, and f maps from ^{n} to . If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = {}_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

In vector notation, this may be equivalently written as

$$\nabla \; z = \quad \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \quad ^{\top} \nabla \; z, \tag{6.46}$$

where $\frac{\partial}{\partial}$ is the $n \times m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial}{\partial}$ by a gradient $\nabla \; z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Usually we apply the back-propagation algorithm to tensors of arbitrary dimensionality, not merely to vectors. Conceptually, this is exactly the same as back-propagation with vectors. The only difference is how the numbers are arranged in a grid to form a tensor. We could imagine flattening each tensor into a vector before we run back-propagation, computing a vector-valued gradient, and then reshaping the gradient back into a tensor. In this rearranged view, back-propagation is still just multiplying Jacobians by gradients.

To denote the gradient of a value z with respect to a tensor , we write $\nabla \; z$, just as if were a vector. The indices into now have multiple coordinates—for example, a 3-D tensor is indexed by three coordinates. We can abstract this away by using a single variable i to represent the complete tuple of indices. For all possible index tuples i, $(\nabla \; z)_i$ gives $\frac{\partial z}{\partial}$. This is exactly the same as how for all possible integer indices i into a vector, $(\nabla \; z)_i$ gives $\frac{\partial z}{\partial x}$. Using this notation, we can write the chain rule as it applies to tensors. If $= g(\;)$ and $z = f(\;)$, then

$$\nabla \; z = {}_j (\nabla \quad {}_j) \frac{\partial z}{\partial \;\; {}_j}. \tag{6.47}$$

Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar. Actually evaluating that expression in a computer, however, introduces some extra considerations.

Specifically, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient will need to choose whether to store these subexpressions or to recompute them several times. An example of how these repeated subexpressions arise is given in figure 6.9. In some cases, computing the same subexpression twice would simply be wasteful. For complicated graphs, there can be exponentially many of these wasted computations, making a naive implementation of the chain rule infeasible. In other cases, computing the same subexpression twice could be a valid way to reduce memory consumption at the cost of higher runtime.

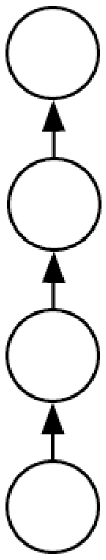
We begin with a version of the back-propagation algorithm that specifies the actual gradient computation directly (algorithm 6.2 along with algorithm 6.1 for the associated forward computation), in the order it will actually be done and according to the recursive application of chain rule. One could either directly perform these computations or view the description of the algorithm as a symbolic specification of the computational graph for computing the back-propagation. However, this formulation does not make explicit the manipulation and the construction of the symbolic graph that performs the gradient computation. Such a formulation is presented in section 6.5.6, with algorithm 6.5, where we also generalize to nodes that contain arbitrary tensors.

First consider a computational graph describing how to compute a single scalar $u^{(n)}$ (say, the loss on a training example). This scalar is the quantity whose gradient we want to obtain, with respect to the n_i input nodes $u^{(1)}$ to $u^{(n\ )}$. In other words, we wish to compute $\frac{\partial u}{\partial u}$ for all $i \in \{1, 2, \ldots, n_i\}$. In the application of back-propagation to computing gradients for gradient descent over parameters, $u^{(n)}$ will be the cost associated with an example or a minibatch, while $u^{(1)}$ to $u^{(n\ )}$ correspond to the parameters of the model.

We will assume that the nodes of the graph have been ordered in such a way that we can compute their output one after the other, starting at $u^{(n\ +1)}$ and going up to $u^{(n)}$. As defined in algorithm 6.1, each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by evaluating the function

$$u^{(i)} = f(\quad {}^{(i)}), \tag{6.48}$$

where ${}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$.



$w \in$ $f :\ \ \rightarrow$

$x = f(w)$ $y = f(x)$ $z = f(y)$

—

$$\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f\ \ (y)f\ (x)f\ \ (w) \\
=&f\ \ (f(f(w)))f\ (f(w))f\ (w).
\end{aligned}$$

$f(w)$

x

$f(w)$ $f(w)$

That algorithm specifies the forward propagation computation, which we could put in a graph $\mathcal{G}$. To perform back-propagation, we can construct a computational graph that depends on $\mathcal{G}$ and adds to it an extra set of nodes. These form a subgraph $\mathcal{B}$ with one node per node of $\mathcal{G}$. Computation in $\mathcal{B}$ proceeds in exactly the reverse of the order of computation in $\mathcal{G}$, and each node of $\mathcal{B}$ computes the derivative $\frac{\partial u}{\partial u}$ associated with the forward graph node $u^{(i)}$. This is done using the chain rule with respect to scalar output $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \quad_{i:j\in Pa(u \quad)} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \tag{6.53}$$

as specified by algorithm 6.2. The subgraph $\mathcal{B}$ contains exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of $\mathcal{G}$. The edge from $u^{(j)}$ to $u^{(i)}$ is associated with the computation of $\frac{\partial u}{\partial u}$. In addition, a dot product is performed for each node, between the gradient already computed with respect to nodes $u^{(i)}$ that are children of $u^{(j)}$ and the vector containing the partial derivatives $\frac{\partial u}{\partial u}$ for the same children nodes $u^{(i)}$. To summarize, the amount of computation required for performing the back-propagation scales linearly with the number of edges in $\mathcal{G}$, where the computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents) as well as performing one multiplication and one addition. Below, we generalize this analysis to tensor-valued nodes, which is just a way to group multiple scalar values in the same node and enable more

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n\ )}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n\ )}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

```
for $i = 1, \ldots, n_i$ do
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ do
  $^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\ ^{(i)})$
end for
return $u^{(n)}$
```

efficient implementations.

The back-propagation algorithm is designed to reduce the number of common subexpressions without regard to memory. Specifically, it performs on the order of one Jacobian product per node in the graph. This can be seen from the fact that backprop (algorithm 6.2) visits each edge from node $u^{(j)}$ to node $u^{(i)}$ of the graph exactly once in order to obtain the associated partial derivative $\frac{\partial u}{\partial u}$. Back-propagation thus avoids the exponential explosion in repeated subexpressions. Other algorithms may be able to avoid more subexpressions by performing simplifications on the computational graph, or may be able to conserve memory by recomputing rather than storing some subexpressions. We revisit these ideas after describing the back-propagation algorithm itself.

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n\ )}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u}{\partial u}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

```
Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.
Initialize , a data structure that will store the derivatives that have been computed. The entry _ [$u^{(i)}$] will store the computed value of $\frac{\partial u}{\partial u}$.
_ [$u^{(n)}$] $\leftarrow 1$
for $j = n - 1$ down to 1 do
  The next line computes $\frac{\partial u}{\partial u} = \sum_{i:j \in Pa(u\ )} \frac{\partial u}{\partial u} \frac{\partial u}{\partial u}$ using stored values:
  _ [$u^{(j)}$] $\leftarrow \sum_{i:j \in Pa(u\ )}$ _ [$u^{(i)}$]$\frac{\partial u}{\partial u}$
end for
return { _ [$u^{(i)}$] | $i = 1, \ldots, n_i$}
```

To clarify the above definition of the back-propagation computation, let us consider the specific graph associated with a fully-connected multi layer MLP.

Algorithm 6.3 first shows the forward propagation, which maps parameters to the supervised loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ associated with a single (input,target) training example $(\boldsymbol{x}, \boldsymbol{y})$, with $\hat{\boldsymbol{y}}$ the output of the neural network when $\boldsymbol{x}$ is provided in input.

Algorithm 6.4 then shows the corresponding computation to be done for applying the back-propagation algorithm to this graph.

Algorithms 6.3 and 6.4 are demonstrations chosen to be simple and straightforward to understand. However, they are specialized to one specific problem.

Modern software implementations are based on the generalized form of back-propagation described in section 6.5.6 below, which can accommodate any computational graph by explicitly manipulating a data structure for representing symbolic computation.

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

```
Require: Network depth, $l$
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \ldots, l$ do
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$
```

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses, in addition to the input $\boldsymbol{x}$, a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

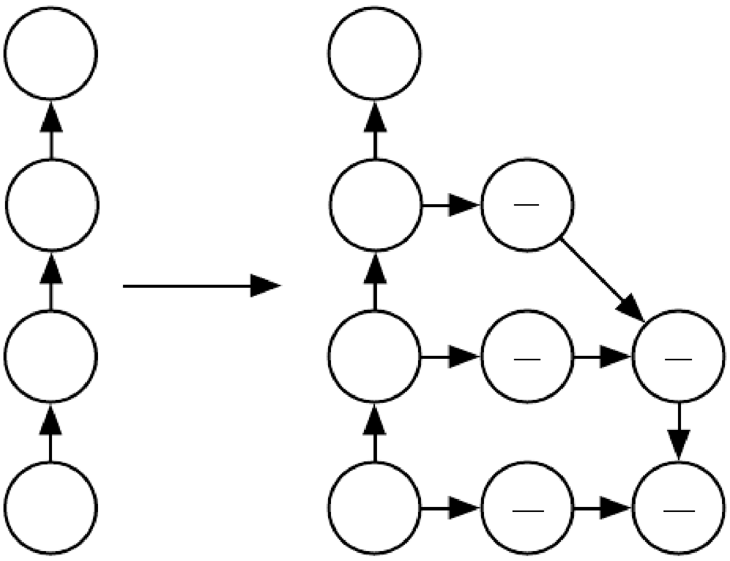
```
After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{}} J = \nabla_{\hat{}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ do
  Convert the gradient on the layer's output into a gradient on the pre-
  nonlinearity activation (element-wise multiplication if $f$ is element-wise):

  $\boldsymbol{g} \leftarrow \nabla \quad J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including the regularization term,
  where needed):
  $\nabla \quad J = \boldsymbol{g} + \lambda \nabla \quad \Omega(\theta)$
  $\nabla \quad J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla \quad \Omega(\theta)$
  Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla \quad J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$
end for
```

Algebraic expressions and computational graphs both operate on **symbols**, or variables that do not have specific values. These algebraic and graph-based representations are called **symbolic representations**. When we actually use or train a neural network, we must assign specific values to these symbols. We replace a symbolic input to the network $\boldsymbol{x}$ with a specific **numeric** value, such as $[1.2, 3.765, -1.8]^\top$.

Some approaches to back-propagation take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values. We call this approach **symbol-to-number differentiation**. This is the approach used by libraries such as Torch (Collobert *et al.*, 2011b) and Caffe (Jia, 2013).

Another approach is to take a computational graph and add additional nodes



$z = f(f(f(w)))$

—

to the graph that provide a symbolic description of the desired derivatives. This is the approach taken by Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) and TensorFlow (Abadi *et al.*, 2015). An example of how it works is illustrated in figure 6.10. The primary advantage of this approach is that the derivatives are described in the same language as the original expression. Because the derivatives are just another computational graph, it is possible to run back-propagation again, differentiating the derivatives to obtain higher derivatives. (Computation of higher-order derivatives is described in section 6.5.10.)

We will use the latter approach and describe the back-propagation algorithm in terms of constructing a computational graph for the derivatives. Any subset of the graph may then be evaluated using specific numerical values at a later time. This allows us to avoid specifying exactly when each operation should be computed. Instead, a generic graph evaluation engine can evaluate every node as soon as its parents' values are available.

The description of the symbol-to-symbol based approach subsumes the symbol-to-number approach. The symbol-to-number approach can be understood as performing exactly the same computations as are done in the graph built by the symbol-to-symbol approach. The key difference is that the symbol-to-number approach does not expose the graph.

The back-propagation algorithm is very simple. To compute the gradient of some scalar z with respect to one of its ancestors $\boldsymbol{x}$ in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z. We continue multiplying by Jacobians, traveling backward through the graph in this way until we reach $\boldsymbol{x}$. For any node that may be reached by going backward from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

More formally, each node in the graph $\mathcal{G}$ corresponds to a variable. To achieve maximum generality, we describe this variable as being a tensor . Tensors in general can have any number of dimensions. They subsume scalars, vectors, and matrices.

We assume that each variable is associated with the following subroutines:

- _ ( ): This returns the operation that computes , represented by the edges coming into in the computational graph. For example, there may be a Python or C++ class representing the matrix multiplication operation, and the function. Suppose we have a variable that is created by matrix multiplication, $\boldsymbol{C} = \boldsymbol{AB}$. Then _ ( ) returns a pointer to an instance of the corresponding C++ class.

- _ ($,\mathcal{G}$): This returns the list of variables that are children of in the computational graph $\mathcal{G}$.

- _ ($,\mathcal{G}$) : This returns the list of variables that are parents of in the computational graph $\mathcal{G}$.

Each operation is also associated with a operation. This operation can compute a Jacobian-vector product as described by equation 6.47. This is how the back-propagation algorithm is able to achieve great generality. Each operation is responsible for knowing how to back-propagate through the edges in the graph that it participates in. For example, we might use a matrix multiplication operation to create a variable $\boldsymbol{C} = \boldsymbol{AB}$. Suppose that the gradient of a scalar z with respect to $\boldsymbol{C}$ is given by $\boldsymbol{G}$. The matrix multiplication operation is responsible for defining two back-propagation rules, one for each of its input arguments. If we call the method to request the gradient with respect to $\boldsymbol{A}$ given that the gradient on the output is $\boldsymbol{G}$, then the method of the matrix multiplication operation must state that the gradient with respect to $\boldsymbol{A}$ is given by $\boldsymbol{GB}^\top$. Likewise, if we call the method to request the gradient with respect to $\boldsymbol{B}$, then the matrix operation is responsible for implementing the method and specifying that the desired gradient is given by $\boldsymbol{A}^\top\boldsymbol{G}$. The back-propagation algorithm itself does not need to know any differentiation rules. It only needs to call each operation's rules with the right arguments. Formally, . (, ,) must return

$$\underset{i}{} (\nabla \quad . (\quad) \quad {}_i) \quad {}_i, \tag{6.54}$$

which is just an implementation of the chain rule as expressed in equation 6.47. Here, is a list of inputs that are supplied to the operation, is the mathematical function that the operation implements, is the input whose gradient we wish to compute, and is the gradient on the output of the operation.

The method should always pretend that all its inputs are distinct from each other, even if they are not. For example, if the operator is passed two copies of x to compute x^2, the method should still return x as the derivative with respect to both inputs. The back-propagation algorithm will later add both of these arguments together to obtain $2x$, which is the correct total derivative on x.

Software implementations of back-propagation usually provide both the operations and their methods, so that users of deep learning software libraries are able to back-propagate through graphs built using common operations like matrix multiplication, exponents, logarithms, and so on. Software engineers who build a new implementation of back-propagation or advanced users who need to add their own operation to an existing library must usually derive the method for any new operations manually.

The back-propagation algorithm is formally described in algorithm 6.5.

In section 6.5.2, we explained that back-propagation was developed in order to avoid computing the same subexpression in the chain rule multiple times. The naive algorithm could have exponential runtime due to these repeated subexpressions. Now that we have specified the back-propagation algorithm, we can understand its computational cost. If we assume that each operation evaluation has roughly the

Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the subroutine of algorithm 6.6

.

```
Require: , the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: $z$, the variable to be differentiated
  Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of $z$ and descendents
  of nodes in .
  Initialize , a data structure associating tensors to their gradients
  _ $[z] \leftarrow 1$
  for in do
    _ $( , \mathcal{G}, \mathcal{G} ',$ _ $)$
  end for
  Return restricted to
```

Algorithm 6.6 The inner loop subroutine _ $(, \mathcal{G}, \mathcal{G}',$ _ $)$ of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

```
Require: , the variable whose gradient should be added to $\mathcal{G}$ and
Require: $\mathcal{G}$, the graph to modify
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient
Require: , a data structure mapping nodes to their gradients
  if is in then
    Return _ [ ]
  end if
  $i \leftarrow 1$
  for in _ $( , \mathcal{G} ')$ do
    $\leftarrow$ _ $( )$
    $\leftarrow$ _ $( , \mathcal{G}, \mathcal{G} ',$ _ $)$
    $^{(i)} \leftarrow$ . $($ _ $( , \mathcal{G} '), , )$
    $i \leftarrow i + 1$
  end for
  $\leftarrow \sum_i {}^{(i)}$
  _ $[ ] =$
  Insert and the operations creating it into $\mathcal{G}$
  Return
```

same cost, then we may analyze the computational cost in terms of the number of operations executed. Keep in mind here that we refer to an operation as the fundamental unit of our computational graph, which might actually consist of several arithmetic operations (for example, we might have a graph that treats matrix multiplication as a single operation). Computing a gradient in a graph with n nodes will never execute more than $O(n^2)$ operations or store the output of more than $O(n^2)$ operations. Here we are counting operations in the computational graph, not individual operations executed by the underlying hardware, so it is important to remember that the runtime of each operation may be highly variable. For example, multiplying two matrices that each contain millions of entries might correspond to a single operation in the graph. We can see that computing the gradient requires at most $O(n^2)$ operations because the forward propagation stage will at worst execute all n nodes in the original graph (depending on which values we want to compute, we may not need to execute the entire graph). The back-propagation algorithm adds one Jacobian-vector product, which should be expressed with $O(1)$ nodes, per edge in the original graph. Because the computational graph is a directed acyclic graph it has at most $O(n^2)$ edges. For the kinds of graphs that are commonly used in practice, the situation is even better. Most neural network cost functions are roughly chain-structured, causing back-propagation to have $O(n)$ cost. This is far better than the naive approach, which might need to execute exponentially many nodes. This potentially exponential cost can be seen by expanding and rewriting the recursive chain rule (equation 6.53) nonrecursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \underset{\substack{\text{path}\,(u\quad,u\quad,\ldots,u\quad),\\ \text{from}\,\pi\quad=j\,\text{to}\,\pi\quad=n}}{} \quad \overset{t}{\underset{k=2}{}} \frac{\partial u^{(\pi\quad)}}{\partial u^{(\pi\quad)}}. \tag{6.55}$$

Since the number of paths from node j to node n can grow exponentially in the length of these paths, the number of terms in the above sum, which is the number of such paths, can grow exponentially with the depth of the forward propagation graph. This large cost would be incurred because the same computation for $\frac{\partial u}{\partial u}$ would be redone many times. To avoid such recomputation, we can think of back-propagation as a table-filling algorithm that takes advantage of storing intermediate results $\frac{\partial u}{\partial u}$. Each node in the graph has a corresponding slot in a table to store the gradient for that node. By filling in these table entries in order, back-propagation avoids repeating many common subexpressions. This table-filling strategy is sometimes called **dynamic programming**.

As an example, we walk through the back-propagation algorithm as it is used to train a multilayer perceptron.

Here we develop a very simple multilayer perceptron with a single hidden layer. To train this model, we will use minibatch stochastic gradient descent. The back-propagation algorithm is used to compute the gradient of the cost on a single minibatch. Specifically, we use a minibatch of examples from the training set formatted as a design matrix $\boldsymbol{X}$ and a vector of associated class labels $\boldsymbol{y}$. The network computes a layer of hidden features $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$. To simplify the presentation we do not use biases in this model. We assume that our graph language includes a operation that can compute $\max\{0, \boldsymbol{Z}\}$ element-wise. The predictions of the unnormalized log probabilities over classes are then given by $\boldsymbol{H}\boldsymbol{W}^{(2)}$. We assume that our graph language includes a operation that computes the cross-entropy between the targets $\boldsymbol{y}$ and the probability distribution defined by these unnormalized log probabilities. The resulting cross-entropy defines the cost J_{MLE}. Minimizing this cross-entropy performs maximum likelihood estimation of the classifier. However, to make this example more realistic, we also include a regularization term. The total cost

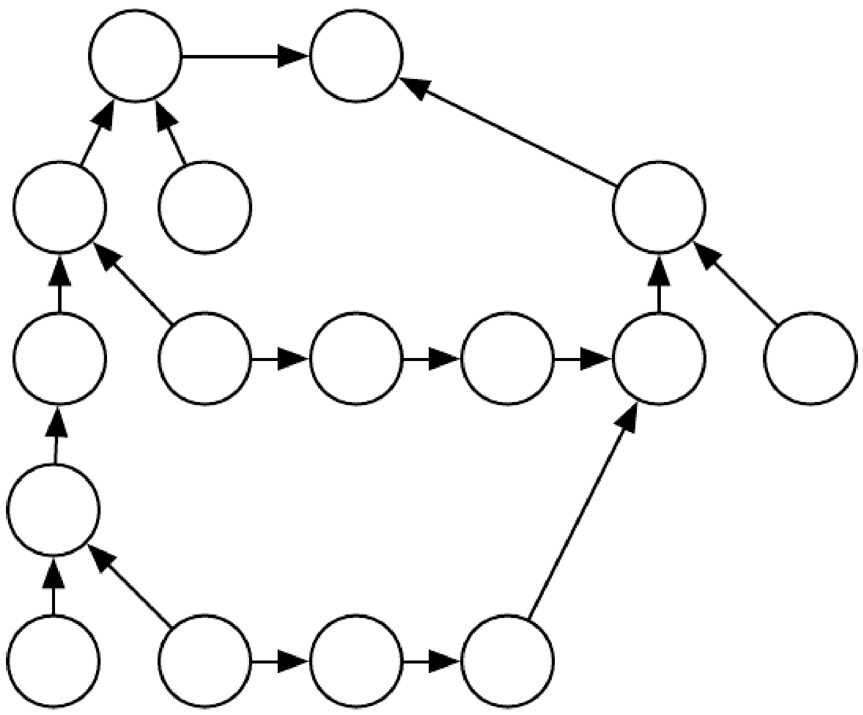
$$J = J_{\mathrm{MLE}} + \lambda \quad \mathop{}_{i,j} \quad W_{i,j}^{(1)} \quad {}^{2} + \quad \mathop{}_{i,j} \quad W_{i,j}^{(2)} \quad {}^{2} \tag{6.56}$$

consists of the cross-entropy and a weight decay term with coefficient λ. The computational graph is illustrated in figure 6.11.

The computational graph for the gradient of this example is large enough that it would be tedious to draw or to read. This demonstrates one of the benefits of the back-propagation algorithm, which is that it can automatically generate gradients that would be straightforward but tedious for a software engineer to derive manually.

We can roughly trace out the behavior of the back-propagation algorithm by looking at the forward propagation graph in figure 6.11. To train, we wish to compute both $\nabla \quad J$ and $\nabla \quad J$. There are two different paths leading backward from J to the weights: one through the cross-entropy cost, and one through the weight decay cost. The weight decay cost is relatively simple; it will always contribute $2\lambda\boldsymbol{W}^{(i)}$ to the gradient on $\boldsymbol{W}^{(i)}$.

The other path through the cross-entropy cost is slightly more complicated. Let $\boldsymbol{G}$ be the gradient on the unnormalized log probabilities $\boldsymbol{U}^{(2)}$ provided by the operation. The back-propagation algorithm now needs to



explore two different branches. On the shorter branch, it adds $\boldsymbol{H}^\top \boldsymbol{G}$ to the gradient on $\boldsymbol{W}^{(2)}$, using the back-propagation rule for the second argument to the matrix multiplication operation. The other branch corresponds to the longer chain descending further along the network. First, the back-propagation algorithm computes $\nabla \quad J = \boldsymbol{G}\boldsymbol{W}^{(2)\top}$ using the back-propagation rule for the first argument to the matrix multiplication operation. Next, the operation uses its back-propagation rule to zero out components of the gradient corresponding to entries of $\boldsymbol{U}^{(1)}$ that are less than 0. Let the result be called $\boldsymbol{G}'$. The last step of the back-propagation algorithm is to use the back-propagation rule for the second argument of the operation to add $\boldsymbol{X}^\top \boldsymbol{G}'$ to the gradient on $\boldsymbol{W}^{(1)}$.

After these gradients have been computed, the gradient descent algorithm, or another optimization algorithm, uses these gradients to update the parameters.

For the MLP, the computational cost is dominated by the cost of matrix multiplication. During the forward propagation stage, we multiply by each weight matrix, resulting in $O(w)$ multiply-adds, where w is the number of weights. During the backward propagation stage, we multiply by the transpose of each weight matrix, which has the same computational cost. The main memory cost of the algorithm is that we need to store the input to the nonlinearity of the hidden layer.

This value is stored from the time it is computed until the backward pass has returned to the same point. The memory cost is thus $O(mn_h)$, where m is the number of examples in the minibatch and n_h is the number of hidden units.

Our description of the back-propagation algorithm here is simpler than the implementations actually used in practice.

As noted above, we have restricted the definition of an operation to be a function that returns a single tensor. Most software implementations need to support operations that can return more than one tensor. For example, if we wish to compute both the maximum value in a tensor and the index of that value, it is best to compute both in a single pass through memory, so it is most efficient to implement this procedure as a single operation with two outputs.

We have not described how to control the memory consumption of back-propagation. Back-propagation often involves summation of many tensors together. In the naive approach, each of these tensors would be computed separately, then all of them would be added in a second step. The naive approach has an overly high memory bottleneck that can be avoided by maintaining a single buffer and adding each value to that buffer as it is computed.

Real-world implementations of back-propagation also need to handle various data types, such as 32-bit floating point, 64-bit floating point, and integer values. The policy for handling each of these types takes special care to design.

Some operations have undefined gradients, and it is important to track these cases and determine whether the gradient requested by the user is undefined.

Various other technicalities make real-world differentiation more complicated. These technicalities are not insurmountable, and this chapter has described the key intellectual tools needed to compute derivatives, but it is important to be aware that many more subtleties exist.

The deep learning community has been somewhat isolated from the broader computer science community and has largely developed its own cultural attitudes concerning how to perform differentiation. More generally, the field of **automatic differentiation** is concerned with how to compute derivatives algorithmically. The back-propagation algorithm described here is only one approach to automatic differentiation. It is a special case of a broader class of techniques called **reverse**

mode accumulation. Other approaches evaluate the subexpressions of the chain rule in different orders. In general, determining the order of evaluation that results in the lowest computational cost is a difficult problem. Finding the optimal sequence of operations to compute the gradient is NP-complete (Naumann, 2008), in the sense that it may require simplifying algebraic expressions into their least expensive form.

For example, suppose we have variables $p_1, p_2, \ldots, p_n$ representing probabilities, and variables $z_1, z_2, \ldots, z_n$ representing unnormalized log probabilities. Suppose we define

$$q_i = \frac{\exp(z_i)}{{}_i \exp(z_i)}, \tag{6.57}$$

where we build the softmax function out of exponentiation, summation and division operations, and construct a cross-entropy loss $J = - \quad {}_i p_i \log q_i$. A human mathematician can observe that the derivative of J with respect to z_i takes a very simple form: $q_i - p_i$. The back-propagation algorithm is not capable of simplifying the gradient this way and will instead explicitly propagate gradients through all the logarithm and exponentiation operations in the original graph. Some software libraries such as Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) are able to perform some kinds of algebraic substitution to improve over the graph proposed by the pure back-propagation algorithm.

When the forward graph $\mathcal{G}$ has a single output node and each partial derivative $\frac{\partial u}{\partial u}$ can be computed with a constant amount of computation, back-propagation guarantees that the number of computations for the gradient computation is of the same order as the number of computations for the forward computation: this can be seen in algorithm 6.2, because each local partial derivative $\frac{\partial u}{\partial u}$ needs to be computed only once along with an associated multiplication and addition for the recursive chain-rule formulation (equation 6.53). The overall computation is therefore $O(\#\text{ edges})$. It can potentially be reduced, however, by simplifying the computational graph constructed by back-propagation, and this is an NP-complete task. Implementations such as Theano and TensorFlow use heuristics based on matching known simplification patterns to iteratively attempt to simplify the graph. We defined back-propagation only for computing a scalar output gradient, but back-propagation can be extended to compute a Jacobian (either of k different scalar nodes in the graph, or of a tensor-valued node containing k values). A naive implementation may then need k times more computation: for each scalar internal node in the original forward graph, the naive implementation computes k gradients instead of a single gradient. When the number of outputs of the graph is larger than the number of inputs, it is sometimes preferable to use another form of automatic differentiation called **forward mode accumulation**. Forward mode accumulation has been proposed for obtaining real-time computation of gradients in recurrent networks, for example (Williams and Zipser, 1989). This approach also avoids the need to store the values and gradients for the whole graph, trading off computational efficiency for memory. The relationship between forward mode and backward mode is analogous to the relationship between left-multiplying versus right-multiplying a sequence of matrices, such as

$$\boldsymbol{ABCD}, \tag{6.58}$$

where the matrices can be thought of as Jacobian. For example, if $\boldsymbol{D}$ is a column vector while $\boldsymbol{A}$ has many rows, the graph will have a single output and many inputs, and starting the multiplications from the end and going backward requires only matrix-vector products. This order corresponds to the backward mode. Instead, starting to multiply from the left would involve a series of matrix-matrix products, which makes the whole computation much more expensive. If $\boldsymbol{A}$ has fewer rows than $\boldsymbol{D}$ has columns, however, it is cheaper to run the multiplications left-to-right, corresponding to the forward mode.

In many communities outside machine learning, it is more common to implement differentiation software that acts directly on traditional programming language code, such as Python or C code, and automatically generates programs that differentiate functions written in these languages. In the deep learning community, computational graphs are usually represented by explicit data structures created by specialized libraries. The specialized approach has the drawback of requiring the library developer to define the methods for every operation and limiting the user of the library to only those operations that have been defined. Yet the specialized approach also has the benefit of allowing customized back-propagation rules to be developed for each operation, enabling the developer to improve speed or stability in nonobvious ways that an automatic procedure would presumably be unable to replicate.

Back-propagation is therefore not the only way or the optimal way of computing the gradient, but it is a practical method that continues to serve the deep learning community well. In the future, differentiation technology for deep networks may improve as deep learning practitioners become more aware of advances in the broader field of automatic differentiation.

Some software frameworks support the use of higher-order derivatives. Among the deep learning software frameworks, this includes at least Theano and TensorFlow.

These libraries use the same kind of data structure to describe the expressions for derivatives as they use to describe the original function being differentiated. This means that the symbolic differentiation machinery can be applied to derivatives.

In the context of deep learning, it is rare to compute a single second derivative of a scalar function. Instead, we are usually interested in properties of the Hessian matrix. If we have a function $f : \ ^n \to \ $, then the Hessian matrix is of size $n \times n$. In typical deep learning applications, n will be the number of parameters in the model, which could easily number in the billions. The entire Hessian matrix is thus infeasible to even represent.

Instead of explicitly computing the Hessian, the typical deep learning approach is to use **Krylov methods**. Krylov methods are a set of iterative techniques for performing various operations, such as approximately inverting a matrix or finding approximations to its eigenvectors or eigenvalues, without using any operation other than matrix-vector products.

To use Krylov methods on the Hessian, we only need to be able to compute the product between the Hessian matrix $\boldsymbol{H}$ and an arbitrary vector $\boldsymbol{v}$. A straightforward technique (Christianson, 1992) for doing so is to compute

$$\boldsymbol{H}\boldsymbol{v} = \nabla \quad (\nabla \ f(x))^\top \boldsymbol{v} \quad . \tag{6.59}$$

Both gradient computations in this expression may be computed automatically by the appropriate software library. Note that the outer gradient expression takes the gradient of a function of the inner gradient expression.

If $\boldsymbol{v}$ is itself a vector produced by a computational graph, it is important to specify that the automatic differentiation software should not differentiate through the graph that produced $\boldsymbol{v}$.

While computing the Hessian is usually not advisable, it is possible to do with Hessian vector products. One simply computes $\boldsymbol{H}\boldsymbol{e}^{(i)}$ for all $i = 1, \ldots, n$, where $\boldsymbol{e}^{(i)}$ is the one-hot vector with $e_i^{(i)} = 1$ and all other entries are equal to 0.

Feedforward networks can be seen as efficient nonlinear function approximators based on using gradient descent to minimize the error in a function approximation. From this point of view, the modern feedforward network is the culmination of centuries of progress on the general function approximation task.

The chain rule that underlies the back-propagation algorithm was invented in the seventeenth century (Leibniz, 1676; L'Hôpital, 1696). Calculus and algebra have long been used to solve optimization problems in closed form, but gradient descent was not introduced as a technique for iteratively approximating the solution to optimization problems until the nineteenth century (Cauchy, 1847).

Beginning in the 1940s, these function approximation techniques were used to motivate machine learning models such as the perceptron. However, the earliest models were based on linear models. Critics including Marvin Minsky pointed out several of the flaws of the linear model family, such as its inability to learn the XOR function, which led to a backlash against the entire neural network approach.

Learning nonlinear functions required the development of a multilayer perceptron and a means of computing the gradient through such a model. Efficient applications of the chain rule based on dynamic programming began to appear in the 1960s and 1970s, mostly for control applications (Kelley, 1960; Bryson and Denham, 1961; Dreyfus, 1962; Bryson and Ho, 1969; Dreyfus, 1973) but also for sensitivity analysis (Linnainmaa, 1976). Werbos (1981) proposed applying these techniques to training artificial neural networks. The idea was finally developed in practice after being independently rediscovered in different ways (LeCun, 1985; Parker, 1985; Rumelhart *et al.*, 1986a). The book **Parallel Distributed Processing** presented the results of some of the first successful experiments with back-propagation in a chapter (Rumelhart *et al.*, 1986b) that contributed greatly to the popularization of back-propagation and initiated a very active period of research in multilayer neural networks. The ideas put forward by the authors of that book, particularly by Rumelhart and Hinton, go much beyond back-propagation. They include crucial ideas about the possible computational implementation of several central aspects of cognition and learning, which came under the name "connectionism" because of the importance this school of thought places on the connections between neurons as the locus of learning and memory. In particular, these ideas include the notion of distributed representation (Hinton *et al.*, 1986).

Following the success of back-propagation, neural network research gained popularity and reached a peak in the early 1990s. Afterwards, other machine learning techniques became more popular until the modern deep learning renaissance that began in 2006.

The core ideas behind modern feedforward networks have not changed substantially since the 1980s. The same back-propagation algorithm and the same approaches to gradient descent are still in use. Most of the improvement in neural network performance from 1986 to 2015 can be attributed to two factors. First, larger datasets have reduced the degree to which statistical generalization is a challenge for neural networks. Second, neural networks have become much larger, because of more powerful computers and better software infrastructure. A small number of algorithmic changes have also improved the performance of neural networks noticeably.

One of these algorithmic changes was the replacement of mean squared error with the cross-entropy family of loss functions. Mean squared error was popular in the 1980s and 1990s but was gradually replaced by cross-entropy losses and the principle of maximum likelihood as ideas spread between the statistics community and the machine learning community. The use of cross-entropy losses greatly improved the performance of models with sigmoid and softmax outputs, which had previously suffered from saturation and slow learning when using the mean squared error loss.

The other major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with piecewise linear hidden units, such as rectified linear units. Rectification using the $\max\{0, z\}$ function was introduced in early neural network models and dates back at least as far as the cognitron and neocognitron (Fukushima, 1975, 1980). These early models did not use rectified linear units but instead applied rectification to nonlinear functions. Despite the early popularity of rectification, it was largely replaced by sigmoids in the 1980s, perhaps because sigmoids perform better when neural networks are very small. As of the early 2000s, rectified linear units were avoided because of a somewhat superstitious belief that activation functions with nondifferentiable points must be avoided. This began to change in about 2009. Jarrett *et al.* (2009) observed that "using a rectifying nonlinearity is the single most important factor in improving the performance of a recognition system," among several different factors of neural network architecture design.

For small datasets, Jarrett *et al.* (2009) observed that using rectifying nonlinearities is even more important than learning the weights of the hidden layers. Random weights are sufficient to propagate useful information through a rectified linear network, enabling the classifier layer at the top to learn how to map different feature vectors to class identities.

When more data is available, learning begins to extract enough useful knowledge to exceed the performance of randomly chosen parameters. Glorot *et al.* (2011a) showed that learning is far easier in deep rectified linear networks than in deep networks that have curvature or two-sided saturation in their activation functions.

Rectified linear units are also of historical interest because they show that neuroscience has continued to have an influence on the development of deep learning algorithms. Glorot *et al.* (2011a) motivated rectified linear units from biological considerations. The half-rectifying nonlinearity was intended to capture these properties of biological neurons: (1) For some inputs, biological neurons are completely inactive. (2) For some inputs, a biological neuron's output is proportional to its input. (3) Most of the time, biological neurons operate in the regime where they are inactive (i.e., they should have **sparse activations**).

When the modern resurgence of deep learning began in 2006, feedforward networks continued to have a bad reputation. From about 2006 to 2012, it was widely believed that feedforward networks would not perform well unless they were assisted by other models, such as probabilistic models. Today, it is now known that with the right resources and engineering practices, feedforward networks perform very well. Today, gradient-based learning in feedforward networks is used as a tool to develop probabilistic models, such as the variational autoencoder and generative adversarial networks, described in chapter 20. Rather than being viewed as an unreliable technology that must be supported by other techniques, gradient-based learning in feedforward networks has been viewed since 2012 as a powerful technology that can be applied to many other machine learning tasks. In 2006, the community used unsupervised learning to support supervised learning, and now, ironically, it is more common to use supervised learning to support unsupervised learning.

Feedforward networks continue to have unfulfilled potential. In the future, we expect they will be applied to many more tasks, and that advances in optimization algorithms and model design will improve their performance even further. This chapter has primarily described the neural network family of models. In the subsequent chapters, we turn to how to use these models—how to regularize and train them.