3 Mutilayer Network Training

Objective

This chapter focuses on the fundamentals of training multilayer networks, and, in particular, deep networks. The training involves choosing a measure of network performance and then determining the network parameters (weights and biases) that optimize that measure of performance. The determination of the optimal parameters is an iterative search process that uses the gradient of performance with respect to the network parameters to decide the direction of search.

Theory and Examples

Much of the material in this chapter is covered in more detail in Chapters 8-13 of NND2. Here we review the basic concepts and discuss specific topics that are especially important for deep networks.

Performance Index

As discussed in Chapter 4 of NND2, there are three types of network training: supervised, unsupervised and reinforcement. In this chapter we concentrate on supervised training, in which examples of network inputs and correct outputs (targets or labels) are provided. In particular, we consider performance learning, in which network weights and biases are adjusted to optimize some measurable quantity that represents network performance. The measure is usually called the *performance index*, or *loss function*.

All supervised learning begins with a data set of network inputs and target outputs:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, ..., \{\mathbf{p}_Q, \mathbf{t}_Q\} \tag{3.1}$$

where Q represents the number of examples in the data set. Each input is applied to the network, and the network output is computed as follows:

$$\mathbf{a}_q^0 = \mathbf{p}_q \tag{3.2}$$

$$\mathbf{a}_q^{m+1} = \mathbf{f}^{m+1}\left(\mathbf{W}^{m+1}\mathbf{a}_q^m + \mathbf{b}^{m+1}\right) \text{ for } m = 0, 1, \ldots, M-1 \tag{3.3}$$

$$\mathbf{a}_q = \mathbf{a}_q^M \tag{3.4}$$

Next, we define the performance index (or loss function). There are a number of potential performance functions, as discussed in Chapter 22 of NND2. We will consider two popular ones – mean square error and cross-entropy. Mean square error is commonly used for function approximation (regression) applications, and cross-entropy is used for classification problems. Mean square error is defined as

$$F(\mathbf{x}) = \frac{1}{QS^M}\sum_{q=1}^{Q}\sum_{i=1}^{S^M}\left(t_{i,q} - a_{i,q}\right)^2 \tag{3.5}$$

where $\mathbf{x}$ is the vector containing all of the weights and biases, S^M is the number of neurons in the output layer (Layer M), $a_{i,q}$ is the i^{th} neuron in the output layer when the q^{th} example input, $\mathbf{p}_q$, is presented, and $t_{i,q}$ is the corresponding element of the target.

Mean square error is a straight-forward description of network performance, and it is minimized when the network outputs exactly match the targets, which produces $F(\mathbf{x}) = 0$. Although mean square error is most-often used for regression problems, it can also be used for classification problems. However, a more commonly used performance function for pattern classification is cross-entropy. This performance function is not quite as intuitive as mean square error, so it is worth taking a few moments to provide some background.

To introduce cross-entropy, we need to view the classification problem from a probabilistic viewpoint. Assume that there exists a conditional probability distribution relating the network input $\mathbf{p}$ to the target output $\mathbf{t}$. In other words, once we know the input, this conditional distribution provides the probabilities associated with any potential target value. The objective of network training is to learn this unknown distribution. If we use the $softmax$ activation function at the final layer of a multilayer network, we can interpret the network as an approximation of this conditional distribution – an input $\mathbf{p}_q$ is applied to the network, and the outputs of the last layer represent the probability that the input belongs to each of the target classes:

$$a_{i,q} = P\left\{\mathbf{p}_q \text{ belongs to class } i\right\} \tag{3.6}$$

Now consider the entire training set of inputs and targets:

$$\mathbf{T} = \begin{bmatrix}\mathbf{t}_1 & \mathbf{t}_2 & \cdots & \mathbf{t}_Q\end{bmatrix}, \mathbf{P} = \begin{bmatrix}\mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_Q\end{bmatrix} \tag{3.7}$$

If the neural network is an accurate estimate of the individual conditional probabilities, then the probability of the training set is:

$$P\left\{\mathbf{T} \mid \mathbf{P}\right\} = \prod_{q=1}^{Q}\prod_{i=1}^{S^M} a_{i,q}^{t_{i,q}} \tag{3.8}$$

where we have assumed that the targets use *one hot encoding* (or one of K encoding), where one element of the target is 1 (corresponding to the correct class) and the other elements are 0. We also assume that each element of the training set is collected independently, which is why we can multiply the individual probabilities.

If the network is going to be trained to best fit the data, then $P\{\mathbf{T} \mid \mathbf{P}\}$ can be thought of as the likelihood function (see Chapter 13 of NND2). We can then adjust the weights and biases to maximize the likelihood. (By maximizing the likelihood, we make the data we collected the most likely data to have occurred.) This is equivalent to minimizing the negative log likelihood, which is the *cross-entropy* performance function:

$$F(\mathbf{x}) = -\sum_{q=1}^{Q}\sum_{i=1}^{S^M} t_{i,q} \ln a_{i,q} \tag{3.9}$$

Optimizing Performance - Gradient Descent

After the performance function (or loss function) has been selected, the next step is to determine the values of the weights and biases that will optimize performance. Before introducing optimization methods, it is appropriate to first analyze the performance function. (See Chapter 8 of NND2 for a more detailed discussion of this material.)

A tool to analyze the performance function is the *Taylor series* expansion. The first three terms of the Taylor series provide a quadratic approximation to the function:

$$F(\mathbf{x}) \cong F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T|_{\mathbf{x}^*}(\mathbf{x}-\mathbf{x}^*) + \frac{1}{2}(\mathbf{x}-\mathbf{x}^*)^T \nabla^2 F(\mathbf{x})|_{\mathbf{x}^*}(\mathbf{x}-\mathbf{x}^*) \tag{3.10}$$

where $\mathbf{x}^*$ is the point where the expansion takes place, $\nabla F(\mathbf{x})$ is the gradient

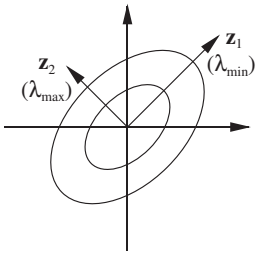
$$\nabla F(\mathbf{x}) = \left[\frac{\partial F(\mathbf{x})}{\partial x_1} \quad \frac{\partial F(\mathbf{x})}{\partial x_2} \quad \ldots \quad \frac{\partial F(\mathbf{x})}{\partial x_n}\right]^T \tag{3.11}$$

and $\nabla^2 F(\mathbf{x})$ is the Hessian

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 F(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 F(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 F(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 F(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 F(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 F(\mathbf{x})}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 F(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 F(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 F(\mathbf{x})}{\partial x_n^2} \end{bmatrix} \tag{3.12}$$

The gradient is a vector that points in the direction in which the performance function is increasing the most rapidly, and the Hessian matrix indicates the curvature of the function. A quadratic function, like the approximation in Eq. 3.10, has elliptical contours

(if the eigenvalues of the Hessian have the same sign), and the major axis of each ellipse is the eigenvector of the Hessian that is associated with the smallest eigenvalue. The eigenvalues of the Hessian represent the curvature of $F(\mathbf{x})$ along the eigenvector directions. This is illustrated in the figure in the margin, where the $\mathbf{z}_i$ are the eigenvectors and the λ_i are the eigenvectors.



The reason that the Taylor series expansion shown in Eq. 3.10 is important is because smooth functions can be closely approximated by these quadratic approximations, at least in small regions. We can use the quadratic approximation, which involves only the gradient and the Hessian, to assist in determining search directions to locate the minimum of $F(\mathbf{x})$.

The basic optimization algorithm takes the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{3.13}$$

where $\mathbf{x}_k$ is the current guess for the optimal weights, $\mathbf{p}_k$ is the search direction and α_k is the step size, or learning rate. Optimization algorithms vary in their choice of search direction and learning rate. (See Chapter 9 of NND2 for a more thorough discussion of optimization algorithms.)

The simplest optimization algorithm is *gradient descent* (GD), or steepest descent, in which the search direction is chosen to be the negative of the gradient direction, which is the direction in which $F(\mathbf{x})$ decreases most rapidly:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x}_k) \tag{3.14}$$

To illustrate the performance of GD, consider the network in Figure 3.1.

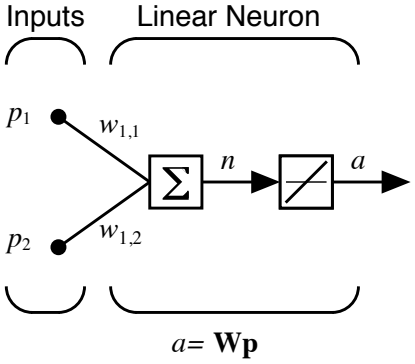


Figure 3.1: Example One Layer Network

To train this network, we use the following data set:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\2\end{bmatrix}, \mathbf{t}_1 = [-1]\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}2\\-1\end{bmatrix}, \mathbf{t}_2 = [-1]\right\} \\ \left\{\mathbf{p}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, \mathbf{t}_3 = [1]\right\}, \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\0\end{bmatrix}, \mathbf{t}_4 = [1]\right\} \tag{3.15}$$

If we then define the following matrices:

$$\mathbf{U} = \begin{bmatrix}\mathbf{p}_1^T\\\mathbf{p}_2^T\\\mathbf{p}_3^T\\\mathbf{p}_4^T\end{bmatrix}, \mathbf{t} = \begin{bmatrix}t_1\\t_2\\t_3\\t_4\end{bmatrix}, \mathbf{x} = \begin{bmatrix}w_{1,1}\\w_{1,2}\end{bmatrix}, \tag{3.16}$$

we can write the sum square error performance function as

$$F(\mathbf{x}) = \sum_{q=1}^{4}\left(t_q - a_q\right)^2 = \left(\mathbf{t} - \mathbf{U}\mathbf{x}\right)^T\left(\mathbf{t} - \mathbf{U}\mathbf{x}\right) \tag{3.17}$$

$$= \left(\mathbf{t}^T\mathbf{t} - 2\mathbf{t}^T\mathbf{U}\mathbf{x} + \mathbf{x}^T\mathbf{U}^T\mathbf{U}\mathbf{x}\right) \tag{3.18}$$

This has the standard form of a quadratic function (see Chapter 8 of NND2):

$$F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} \tag{3.19}$$

$$\tag{3.20}$$

where

$$c = \mathbf{t}^T\mathbf{t}, \mathbf{d} = -2\mathbf{U}^T\mathbf{t}, \mathbf{A} = 2\mathbf{U}^T\mathbf{U} \tag{3.21}$$

The gradient and Hessian of the quadratic function are given by

$$\nabla F(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{d}, \nabla^2 F(\mathbf{x}) = \mathbf{A} \tag{3.22}$$

Plugging in the numbers for the inputs and targets, we find

$$\mathbf{U} = \begin{bmatrix}-1 & 2\\2 & -1\\0 & -1\\-1 & 0\end{bmatrix}, \mathbf{t} = \begin{bmatrix}-1\\-1\\1\\1\end{bmatrix}, \mathbf{A} = \begin{bmatrix}12 & -8\\-8 & 12\end{bmatrix}, \mathbf{d} = \begin{bmatrix}4\\4\end{bmatrix}, c = 4 \tag{3.23}$$

A plot of the function is given in Figure 3.2.

Now let's set up the GD algorithm with a learning rate $\alpha = 0.1$.

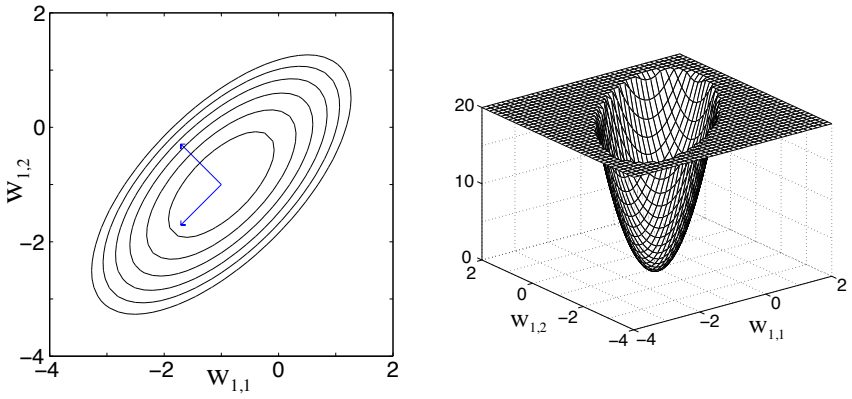
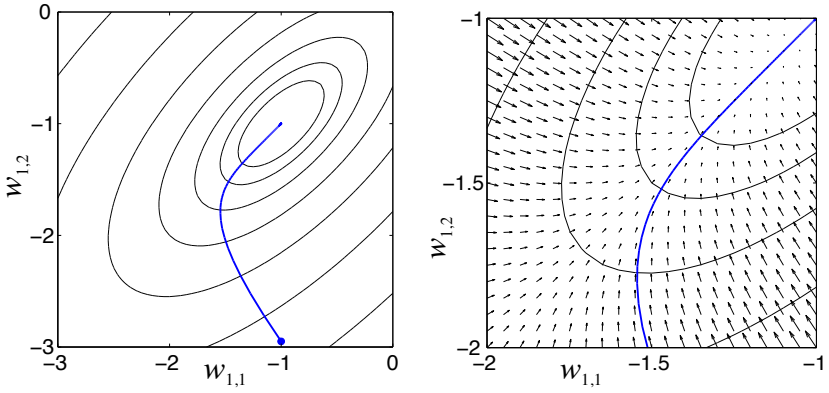


Figure 3.2: Performance Function

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x}_k) = \mathbf{x}_k - 0.01\,(\mathbf{A}\mathbf{x}_k + \mathbf{d}) = \begin{bmatrix} 0.88 & 0.08 \\ 0.08 & 0.88 \end{bmatrix} \mathbf{x}_k + \begin{bmatrix} 0.04 \\ 0.04 \end{bmatrix} \tag{3.24}$$

Figure 3.3 shows the trajectory of the GD algorithm with a very small learning rate. The right side of the figure shows a zoomed plot, with arrows indicating the GD directions. The path of GD is always orthogonal to the contour lines.



(a) Full Trajectory (b) Zoomed Trajectory

Figure 3.3: Gradient Descent Trajectory

When using GD, there is only one parameter to set: the learning rate α. If this is set too small, it may take a long time to converge. If it is set too large, the algorithm can become unstable (see Chapters 9 and 10 of NND2). A standard way around this problem is to start with a large value of α, and then reduce it gradually as the number of iterations increases.

To experiment with the gradient descent algorithm and learning rates, use the Deep Learning Demonstration Gradient Descent (**dl3gd**).

When training deep networks, the data sets are often very large, and it is not practical to compute the gradient of the performance function for the entire data set, which is called *batch* training. Instead, gradients are computed for subsets of the data, which are called *minibatches*. The extreme case of a minibatch occurs when the minibatch contains only a single example. This is called *stochastic gradient descent* (SGD).

For our test problem above, in which a linear network was used, the SGD algorithm would reduce to

$$\hat{F}(\mathbf{x}) = (t_k - a_k)^2 \tag{3.25}$$

$$\nabla \hat{F}(\mathbf{x}) = -2(t_k - a_k)\nabla a_k = -2e_k\mathbf{p}_k \tag{3.26}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e_k \mathbf{p}_k \tag{3.27}$$

Figure 3.4 shows the path of the stochastic gradient algorithm for the test problem. The trajectory of SGD is a noisy version of the GD trajectory. More details on the convergence of SGD can be found in Chapter 10 of NND2.

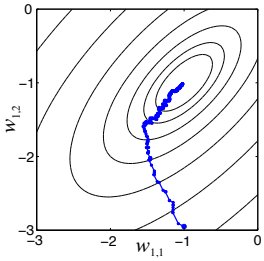


Figure 3.4: Stochastic Gradient Descent Trajectory

The choice of minibatch size depends on the problem, the size of the data set and the memory size of the GPU. It may take some experimentation to select the best minibatch size.

Variations on Gradient Descent – Adam

GD is one of the simplest optimization algorithms, and yet it appears to perform reasonably well on deep networks. However, there are variations of the algorithm that can converge faster. A key drawback of GD is that it only uses the slope of $F(\mathbf{x})$ to determine the search direction. It does not use curvature, which can often be quite helpful. If we consider again the example of the network in Figure 3.1 with the data of Eq. 3.15, the eigenvalues of $\nabla^2 F(\mathbf{x})$ (see Eq. 3.23) are 4 and 20. Because the magnitudes of these eigenvalues (which represent curvatures along the eigenvectors – shown as arrows in Figure 3.2) are so different, the contours, as seen in Figure 3.2, are very elliptical. This means that the initial search directions do not point in the direction of the minimum point, although they point in the steepest downhill direction.

There are a number of optimization algorithms that take curvature into account (see Chapters 9 and 12 of NND2). For example, Newton's method locates the minimum of the quadratic approximation to $F(\mathbf{x})$, shown in Eq. 3.10. Setting the gradient of that approximation to zero yields the following update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \left[\nabla^2 F(\mathbf{x_k})\right]^{-1} \nabla F(\mathbf{x}_k) \tag{3.28}$$

This method will converge in one iteration, if $F(\mathbf{x})$ is quadratic like our earlier example (see Chapter 9 of NND2). However, it requires the calculation of the Hessian matrix and its inverse, which is prohibitive for deep networks. The Levenberg-Marquardt algorithm (see Chapter 12 of NND2) combines an approximation of Newton's method with GD, and is very fast for smaller networks. However, like Newton's method, it requires the inverse of a large matrix and is not practical for deep networks.

Another class of optimization algorithm that takes curvature into account, but which does not require the Hessian calculation, is the conjugate gradient method (see Chapters 9 and 12 of NND2). Conjugate gradient methods use successive gradient calculations to obtain curvature information. They then find a more efficient search direction by modifying the GD direction to account for the curvature of $F(\mathbf{x})$. Conjugate gradient-type methods are well-suited to deep learning, because they account for curvature without the memory requirements and computational burden of algorithms like Newton's method and Levenberg-Marquardt.

There are many optimization algorithms that are inspired by the conjugate gradient concepts, and others will certainly be introduced in the future. To illustrate the genre, we will describe one algorithm that has been popular for deep learning. It is called the *Adam* method [Kingma and Ba, 2014] (derived from *adaptive moment estimation*). Like conjugate gradient algorithms, it modifies the search direction from standard GD using curvature information without the Hessian calculations.

As with the GD algorithm, Adam starts with the gradient $\mathbf{g}_k = \nabla F(\mathbf{x}_k)$. In addition, it computes a smoothed gradient and a smoothed squared gradient:

$$\mathbf{m}_{k+1} = \beta_1 \mathbf{m}_k + (1 - \beta_1)\mathbf{g}_k \tag{3.29}$$

$$\mathbf{v}_{k+1} = \beta_2 \mathbf{v}_k + (1 - \beta_2)\mathbf{g}_k \circ \mathbf{g}_k \tag{3.30}$$

where $\circ$ is the Hadamard (element-by-element) product. These vectors are initialized with zeros: $\mathbf{m}_0 = \mathbf{0}$, $\mathbf{v}_0 = \mathbf{0}$. The smoothed squared gradient $\mathbf{v}_{k+1}$ measures the second moment of the gradient, which is helpful in getting curvature information.

The smoothed values $\mathbf{m}_k$ and $\mathbf{v}_k$ are biased toward zero, because of their initialization. To correct this, a second stage is performed:

$$\hat{\mathbf{m}}_{k+1} = \mathbf{m}_{k+1}/(1-\beta_1^{k+1}) \tag{3.31}$$

$$\hat{\mathbf{v}}_{k+1} = \mathbf{v}_{k+1}/(1-\beta_2^{k+1}) \tag{3.32}$$

where $0 < \beta_1, \beta_2 < 1$ are constant exponential decay rates. The adjusted smoothed gradient $\hat{\mathbf{m}}_{k+1}$ is the search direction, and the adjusted smoothed squared gradient $\hat{\mathbf{v}}_{k+1}$ is used to adjust the learning rate:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \hat{\mathbf{m}}_{k+1} \oslash \left(\sqrt{\hat{\mathbf{v}}_{i+1}} + \epsilon\right) \tag{3.33}$$

where $\oslash$ is the Hadamard (element-by-element) division and ϵ is a small positive constant to prevent division by zero.

The algorithm can be made more numerically efficient by replacing Eq. 3.31-3.33 with the following:

$$\alpha_{k+1} = \alpha(1-\beta_1^{k+1})/\sqrt{1-\beta_2^{k+1}} \tag{3.34}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_{k+1}\mathbf{m}_{k+1} \oslash (\sqrt{\mathbf{v}_{i+1}} + \epsilon) \tag{3.35}$$

In [Kingma and Ba, 2014], they suggest that appropriate default settings for the algorithm parameters are: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

This algorithm has elements of conjugate gradient algorithms as well as momentum algorithms, in which the search direction is a smoothed version of the gradient (see Chapter 12 of NND2). When the gradient is always of the same sign, the smoothed gradient stays large, and the weight adjustment is large. When the gradient is changing sign (e.g., when oscillating across a narrow valley), the weight adjustment gets smaller ($\mathbf{m}_k$ gets smaller because negative and positive values are being averaged, while $\mathbf{v}_k$ stays large because squared values are always positive). This allows us to have a relatively large learning rate, while maintaining stability.

There are other gradient-based algorithms that have been proposed, and there will undoubtedly be more in the future. Our purpose here is to emphasize that the gradient is an important tool

in network training, and that variations of GD can offer faster convergence. The next step is to demonstrate how the gradient can be efficiently computed.

To experiment with the Adam algorithm, use the Deep Learning Demonstration Adam Algorithm (**dl3aa**).

Computing the Gradient

The key to most optimization algorithms that are used in training deep networks is the gradient of the performance with respect to the weights and biases in the network. It is important to be able to compute the gradient efficiently for any architecture of network. In this section we will focus on the gradient calculation for multilayer networks with an arbitrary number of layers. This was covered in Chapter 11 of NND2, but we will describe it here in a way that can extend to networks with more general connections, as will be discussed in a later chapter.

For one-layer networks, like the one in Figure 3.1, it is easy to compute the derivative of the performance with respect to the weights, because the network output is a direct function of the weights. However, in multilayer networks, like Figure 2.7, the network output is only indirectly a function of the weights in the first layer. Therefore, to efficiently compute the full gradient, we need to use the chain rule of calculus.

Assume that we want to take the derivative of the performance function with respect to a weight in layer m. We would use the chain rule in the following way:

$$\frac{\partial F(\mathbf{x})}{\partial w_{i,j}^m} = \frac{\partial F(\mathbf{x})}{\partial n_i^m}\frac{\partial n_i^m}{\partial w_{i,j}^m} = \frac{\partial F(\mathbf{x})}{\partial n_i^m}\frac{\partial (\sum_{l=1}^{S^{m-1}} w_{i,l}^m a_l^{m-1} + b_i^m)}{\partial w_{i,j}^m} \tag{3.36}$$

$$= \frac{\partial F(\mathbf{x})}{\partial n_i^m} a_j^{m-1} \tag{3.37}$$

The key to using the chain rule is the choice of intermediate variable. In this case, we chose n_i^m. The reason is that n_i^m is a linear function of $w_{i,j}^m$, so the derivative of n_i^m with respect to $w_{i,j}^m$ is simply a_j^{m-1}.

Eq. 3.37 can be used to compute the elements of the gradient, but we need to find the *sensitivity*

$$\mathbf{s}^m \triangleq \frac{\partial F(\mathbf{x})}{\partial \mathbf{n}^m} \tag{3.38}$$

With this sensitivity vector, we can compute the elements of the gradient associated with weight matrix $\mathbf{W}^m$ using

$$\frac{\partial F(\mathbf{x})}{\partial \mathbf{W}^m} = \mathbf{s}^m \left(\mathbf{a}^{m-1}\right)^T \tag{3.39}$$

The bias gradient elements are simpler, since the bias computation is just an addition:

$$\frac{\partial F(\mathbf{x})}{\partial \mathbf{b}^m} = \mathbf{s}^m \tag{3.40}$$

It remains to compute the sensitivities $\mathbf{s}^m$. They can be calculated using a series of applications of the chain rule, starting at the last layer, and moving backward through the layers:

$$\mathbf{s}^M \rightarrow \mathbf{s}^{M-1} \rightarrow \cdots \rightarrow \mathbf{s}^1 \tag{3.41}$$

This process is called *backpropagation*. Backpropagation can be performed in stages, as we move across the different components of each layer: weight function, net input function and activation function:

$$\mathbf{z}^{m+1} = \mathbf{W}^{m+1}\mathbf{a}^m \tag{3.42}$$

$$\mathbf{n}^{m+1} = \mathbf{z}^{m+1} + \mathbf{b}^{m+1} \tag{3.43}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}\left(\mathbf{n}^{m+1}\right) \tag{3.44}$$

To propagate the sensitivity from layer $m+1$ to layer m we would use the chain rule as follows:

$$\frac{\partial F(\mathbf{x})}{\partial \mathbf{n}^m} = \frac{\partial \left(\mathbf{a}^m\right)^T}{\partial \mathbf{n}^m} \frac{\partial \left(\mathbf{z}^{m+1}\right)^T}{\partial \mathbf{a}^m} \frac{\partial \left(\mathbf{n}^{m+1}\right)^T}{\partial \mathbf{z}^{m+1}} \frac{\partial F(\mathbf{x})}{\partial \mathbf{n}^{m+1}} \tag{3.45}$$

Let's consider each derivative individually. First, for the transfer function we have

$$\frac{\partial \left(\mathbf{a}^m\right)^T}{\partial \mathbf{n}^m} = \dot{\mathbf{F}}^{\mathbf{m}}\left(\mathbf{n}^m\right) = \begin{bmatrix} \frac{\partial f_1^m(\mathbf{n}^m)}{\partial n_1^m} & \frac{\partial f_2^m(\mathbf{n}^m)}{\partial n_1^m} & \cdots & \frac{\partial f_{S^m}^m(\mathbf{n}^m)}{\partial n_1^m} \\ \frac{\partial f_1^m(\mathbf{n}^m)}{\partial n_2^m} & \frac{\partial f_2^m(\mathbf{n}^m)}{\partial n_2^m} & \cdots & \frac{\partial f_{S^m}^m(\mathbf{n}^m)}{\partial n_2^m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_1^m(\mathbf{n}^m)}{\partial n_{S^m}^m} & \frac{\partial f_2^m(\mathbf{n}^m)}{\partial n_{S^m}^m} & \cdots & \frac{\partial f_{S^m}^m(\mathbf{n}^m)}{\partial n_{S^m}^m} \end{bmatrix}, \tag{3.46}$$

for the weight function we have

$$\frac{\partial \left(\mathbf{z}^{m+1}\right)^T}{\partial \mathbf{a}^m} = \frac{\partial (\mathbf{W}^{m+1}\mathbf{a}^m)^T}{\partial \mathbf{a}^m} = \left(\mathbf{W}^{m+1}\right)^T, \tag{3.47}$$

for the net input function we have

$$\frac{\partial \left(\mathbf{n}^{m+1}\right)^T}{\partial \mathbf{z}^{m+1}} = \frac{\partial (\mathbf{z}^{m+1} + \mathbf{b}^{m+1})^T}{\partial \mathbf{z}^{m+1}} = \mathbf{I} \tag{3.48}$$

Putting these steps together, we can write Eq. 3.45 as

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1} \tag{3.49}$$

The process is illustrated in Figure 3.5 for a three layer network. The expressions above the arrows represent the derivatives across the corresponding components of the layer. The backpropagation process involves multiplying by these derivatives. We will see in a later chapter that this can work for feedforward networks with more complex connections between layers. When you propagate the sensitivities backward across a component of the network, you multiply by the derivative of the component.

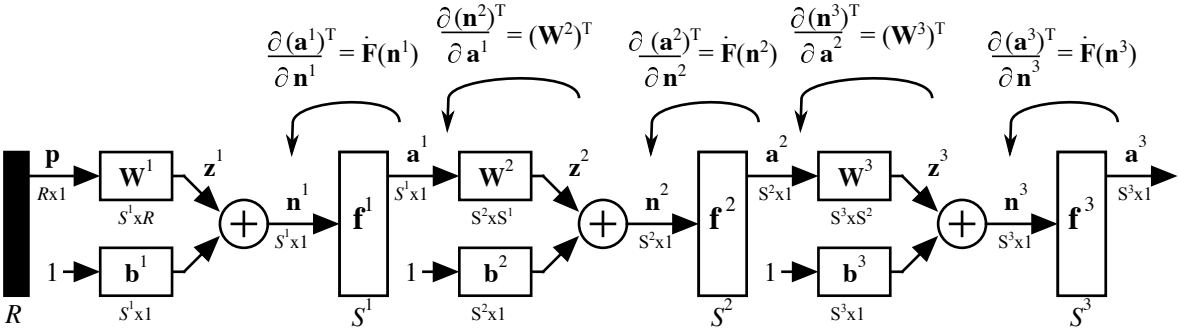


Figure 3.5: Bacpropagation Process

We need to begin the backpropagation recursion of Eq. 3.49 at the final layer of the network – Layer M:

$$\frac{\partial F(\mathbf{x})}{\partial \mathbf{n}^M} = \frac{\partial (\mathbf{a}^M)^T}{\partial \mathbf{n}^M} \frac{\partial F(\mathbf{x})}{\partial \mathbf{a}^M} = \dot{\mathbf{F}}^{\mathbf{M}}\left(\mathbf{n}^M\right) \frac{\partial F(\mathbf{x})}{\partial \mathbf{a}^M} \tag{3.50}$$

For example, if $F(\mathbf{x})$ is mean square error, we have

$$\frac{\partial F(\mathbf{x})}{\partial a_i^M} = \frac{\partial \frac{1}{S^M} \sum_{j=1}^{S^M} \left(t_j - a_j^M\right)^2}{\partial a_i^M} = \frac{-2}{S^M}\left(t_i - a_i^M\right) = \frac{-2}{S^M} e_i \tag{3.51}$$

$$\frac{\partial F(\mathbf{x})}{\partial \mathbf{a}^M} = \frac{-2}{S^M}\mathbf{e} \tag{3.52}$$

where we have considered the stochastic gradient, with one example. We can now write Eq. 3.50 as

$$\mathbf{s}^M = \frac{-2}{S^M}\dot{\mathbf{F}}^{\mathbf{M}}\left(\mathbf{n}^M\right)\mathbf{e} \tag{3.53}$$

The total backpropagation process begins with Eq. 3.53, followed by Eq. 3.49, which is iterated from $m = M - 1$ to $m = 1$. (Eq. 3.53 applies to mean square error performance, but this process can be adjusted for any performance function by substituting the derivative of the chosen performance function into Eq. 3.50.) Once the sensitivities are computed, the parts of the gradient are then computed from Eqs. 3.39 (for the weights) and 3.40 (for the biases).

From this process, the gradient of performance for a single example is obtained (stochastic gradient). Assuming that the total performance function is produced by summing the performance functions for each example (as is the case for mean square error and cross-entropy), the total gradient is obtained by summing the gradients of each example.

Epilogue

The training of deep networks is an optimization problem. The objective is to determine the weights and biases of the network that optimize network performance. We discussed two popular performance functions: mean square error and cross-entropy. Mean square error is used for function fitting (regression), and cross-entropy is used for pattern recognition.

Once the performance function has been chosen, an iterative search is performed to find the weights and biases that optimize performance. The simplest search method is gradient descent, which moves in the direction where the performance function decreases most rapidly. This method uses only the slope of the performance surface, but convergence can be improved by using curvature information. A popular deep learning algorithm that uses curvature information is the Adam method, which was also described in this chapter.

Training algorithms can be run in batch mode, where performance and gradient are computed on the entire data set. They can also be run in stochastic mode, where performance and gradient are computed on each example in the data set separately and weights are updated after each example. There is also an intermediate mode, in which mini-batches of data are used. The choice of the minibatch size is often determined by the size of available GPU memory.

Almost all deep learning algorithms use the gradient in order to determine search directions. This chapter demonstrated how the gradient can be computed for deep multilayer networks. The gradient is computed in stages, moving backward across each stage of the network. The backward operation for each stage involves multiplying by the derivative of the forward operation of that stage. In a later chapter, we will expand the backpropagation algorithm for networks with more general connections between layers.

When training deep networks on practical problems, the objective is not necessarily to optimize performance on the training set, which was our focus in this chapter. Rather, we want to train networks to perform well on unknown, future data. Networks with this ability are said to generalize well (see Chapter 13 of NND2). In the next chapter we will discuss auxiliary procedures that are used in combination with basic training algorithms to improve generalization and otherwise improve training.

Further Reading

[Kingma and Ba, 2014] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014

This is the original description of the Adam algorithm. It also describes AdaMax, which is a variation of Adam that uses the infinity norm.

[Duchi et al., 2011] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011

The Adam algorithm is based, in part, on the AdaGrad algorithm, which is described in this paper. The search direction is modified from the gradient direction using second order information – previous sum of squared gradients.

[Zeiler, 2012] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012

The Adadelta algorithm described in this paper is a modification of the AdaGrad algorithm. It limits the sum of squared previous gradients to a window of fixed size, which prevents the effective learning rate from going to zero.

[Liu and Nocedal, 1989] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989

This limited memory BFGS algorithm is a type of quasi-Newton method. The standard BFGS algorithm requires storing an approximate inverse Hessian matrix, which can be impractically large for deep networks. The L-BFGS algorithm stores a few vectors that represent the inverse Hessian.

Summary of Results

Training Set

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, ..., \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Multilayer Network Forward Operation

$$\begin{aligned} \mathbf{a}_q^0 &= \mathbf{p}_q \\ \mathbf{a}_q^{m+1} &= \mathbf{f}^{m+1}\left(\mathbf{W}^{m+1}\mathbf{a}_q^m + \mathbf{b}^{m+1}\right) \text{ for } m = 0, 1, \ldots, M-1 \\ \mathbf{a}_q &= \mathbf{a}_q^M \end{aligned}$$

Mean Square Error Performance Function

$$F(\mathbf{x}) = \frac{1}{QS^M}\sum_{q=1}^{Q}\sum_{i=1}^{S^M}\left(t_{i,q} - a_{i,q}\right)^2$$

Cross-Entropy Performance Function

$$F(\mathbf{x}) = -\sum_{q=1}^{Q}\sum_{i=1}^{S^M} t_{i,q} \ln a_{i,q}$$

Taylor Series Expansion

$$F(\mathbf{x}) \cong F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T|_{\mathbf{x}^*}\left(\mathbf{x} - \mathbf{x}^*\right) + \frac{1}{2}\left(\mathbf{x} - \mathbf{x}^*\right)^T \nabla^2 F(\mathbf{x})|_{\mathbf{x}^*}\left(\mathbf{x} - \mathbf{x}^*\right)$$

Gradient

$$\nabla F(\mathbf{x}) = \left[\frac{\partial F(\mathbf{x})}{\partial x_1} \quad \frac{\partial F(\mathbf{x})}{\partial x_2} \quad \cdots \quad \frac{\partial F(\mathbf{x})}{\partial x_n}\right]^T$$

Hessian

$$\nabla^2 F(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 F(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 F(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 F(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 F(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 F(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 F(\mathbf{x})}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 F(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 F(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 F(\mathbf{x})}{\partial x_n^2} \end{bmatrix}$$

Basic Optimization Algorithm

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

Gradient Descent

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x}_k)$$

Adam Algorithm

$$\begin{aligned}
\mathbf{g}_k &= \nabla F(\mathbf{x}_k) \\
\mathbf{m}_{k+1} &= \beta_1 \mathbf{m}_k + (1-\beta_1)\mathbf{g}_k \\
\mathbf{v}_{k+1} &= \beta_2 \mathbf{v}_k + (1-\beta_2)\mathbf{g}_k \circ \mathbf{g}_k \\
\alpha_{k+1} &= \alpha(1-\beta_1^{k+1})/\sqrt{1-\beta_2^{k+1}} \\
\mathbf{x}_{k+1} &= \mathbf{x}_k - \alpha_{k+1}\mathbf{m}_{k+1} \oslash (\sqrt{\mathbf{v}_{i+1}} + \epsilon) \\
\alpha &= 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}
\end{aligned}$$

Sensitivity

$$\mathbf{s}^m \triangleq \frac{\partial F(\mathbf{x})}{\partial \mathbf{n}^m}$$

Initialization of Sensitivity

$$\mathbf{s}^M = \dot{\mathbf{F}}^{\mathbf{M}}\left(\mathbf{n}^M\right)\frac{\partial F(\mathbf{x})}{\partial \mathbf{a}^M}$$

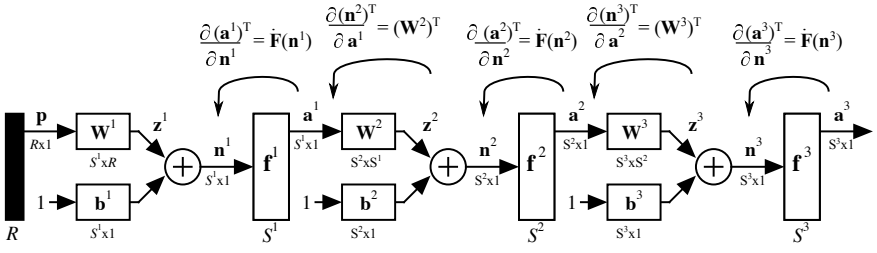
Backpropagation of Sensitivity

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, m = M-1, \ldots, 1$$

Weight Update

$$\begin{aligned}
\frac{\partial F(\mathbf{x})}{\partial \mathbf{W}^m} &= \mathbf{s}^m\left(\mathbf{a}^{m-1}\right)^T \\
\frac{\partial F(\mathbf{x})}{\partial \mathbf{b}^m} &= \mathbf{s}^m
\end{aligned}$$

Backpropagation Process



Solved Problems

P3.1 **The cross-entropy performance function is generally used when the last layer activation function is $softmax$, and the i^{th} output neuron, a_i, is the probability that the input belongs to class i. However, cross-entropy can also be used for a network with a $logsig$ activation function, if there is only one neuron in the last layer. Show how this could be done.**

Using a network with a single $logsig$ output neuron, we can recognize two classes. One class is represented by a target of 1 and the other by a target of 0. The output neuron a represents the probability that the input is in class 1. This means that the probability that the input is in class 0 is $(1 - a)$.

The cross-entropy performance function when there is a $softmax$ activation in the last layer is given by Eq. 3.9:

$$F(\mathbf{x}) = -\sum_{q=1}^{Q}\sum_{i=1}^{S^M} t_{i,q} \ln a_{i,q}$$

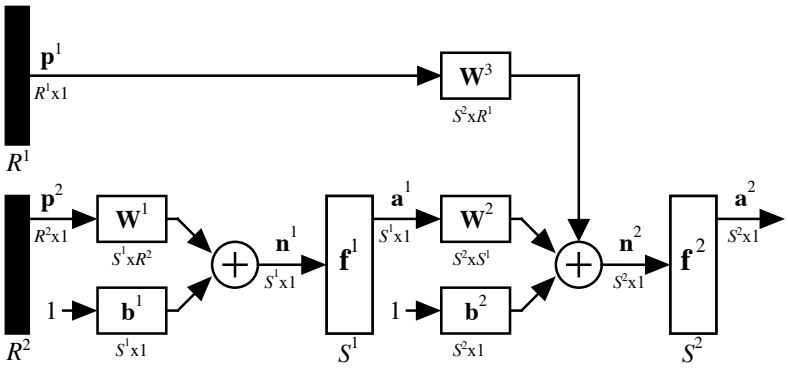
If we had two $softmax$ neurons in the last layer of the network ($S^M = 2$), then we could use this equation. The first output neuron, a_1, represents the probability of class 1, and the second output neuron, a_2, represents the probability of class 0. In a network with one $logsig$ neuron, a represents the probability of class 1, and $(1 - a)$ represents the probability of class 0. The cross-entropy equation then becomes

$$F(\mathbf{x}) = -\sum_{q=1}^{Q} \left[t_q \ln a_q + \left(1 - t_q\right) \ln \left(1 - a_q\right)\right]$$

P3.2 **The backpropagation method can be applied to networks that are not strictly multilayer networks, where there is one input and each layer connects only to the following layer, which produces the backpropagation recursion of Eq. 3.49. How would backpropagation change for the network in the figure below, and how would the portion of the gradient for the weight $\mathbf{W}^3$ be computed?**

The backpropagation process computes the sensitivities

$$\mathbf{s}^m \triangleq \frac{\partial F(\mathbf{x})}{\partial \mathbf{n}^m}$$



For this network, the sensitivities are $\mathbf{s}^2$ and $\mathbf{s}^1$. They would be computed using Eq. 3.50 and Eq. 3.49, as with a standard multilayer network. The additional input connection does not affect the calculation of the sensitivities. The derivative of $\mathbf{n}^2$ with respect to $\mathbf{n}^1$ depends only on $\mathbf{W}^2$ and $\mathbf{f}^1$. Other terms that enter the summation junction are just added, and do not change the effect of $\mathbf{n}^1$ on $\mathbf{n}^2$.

For the elements of the gradient associated with the weight $\mathbf{W}^3$, we first notice that

$$\mathbf{n}^2 = \mathbf{W}^3\mathbf{p}^1 + \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2$$

For element i this would produce

$$n_i^2 = \sum_{j=1}^{R^1} w_{i,j}^3 p_j^1 + \sum_{l=1}^{S^1} w_{i,l}^2 a_l^1 + b_i^2$$

The elements of the gradient associated with $\mathbf{W}^3$ would then be computed as

$$\frac{\partial F(\mathbf{x})}{\partial w_{i,j}^3} = \frac{\partial F(\mathbf{x})}{\partial n_i^2}\frac{\partial n_i^2}{\partial w_{i,j}^3} = s_i^2 \frac{\partial\left(\sum_{j=1}^{R^1} w_{i,j}^3 p_j^1 + \sum_{l=1}^{S^1} w_{i,l}^2 a_l^1 + b_i^m\right)}{\partial w_{i,j}^3}$$
$$= s_i^2 p_j^1$$

P3.3 **Consider again the network in Figure 3.1, and the data set below.**

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-4\\2\end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix}-1\end{bmatrix}\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}1\\2\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}1\end{bmatrix}\right\}$$

i. **Write out an expression for the sum squared error.**

ii. Find the gradient and Hessian of the sum squared error.

iii. Find the eigenvalues and eigenvectors of the Hessian. The eigenvector with smallest eigenvalue will be the major axis of the elliptical contours of the contour plot.

iv. Sketch the contour plot, and sketch the path of the GD algorithm if a very small learning rate is used. Start at the initial weight $\mathbf{W} = \begin{bmatrix} 1 & 1 \end{bmatrix}$.

v. The maximum stable learning rate is 2 divided by the maximum eigenvalue of the Hessian. Find this value. What would the GD path look like for this learning rate?

i. If we define

$$\mathbf{U} = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \end{bmatrix} = \begin{bmatrix} -4 & 2 \\ 1 & 2 \end{bmatrix}, \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} w_{1,1} \\ w_{1,2} \end{bmatrix},$$

We can write the sum square error as

$$\begin{aligned} F(\mathbf{x}) &= \sum_{q=1}^{2} \left(t_q - a_q\right)^2 = \left(\mathbf{t} - \mathbf{U}\mathbf{x}\right)^T \left(\mathbf{t} - \mathbf{U}\mathbf{x}\right) \\ &= \left(\mathbf{t}^T\mathbf{t} - 2\mathbf{t}^T\mathbf{U}\mathbf{x} + \mathbf{x}^T\mathbf{U}^T\mathbf{U}\mathbf{x}\right) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} \end{aligned}$$

Plugging in the numbers, we find

$$c = \mathbf{t}^T\mathbf{t} = 2, \mathbf{d} = -2\mathbf{U}^T\mathbf{t} = \begin{bmatrix} -10 \\ 0 \end{bmatrix}, \mathbf{A} = 2\mathbf{U}^T\mathbf{U} = \begin{bmatrix} 34 & -12 \\ -12 & 16 \end{bmatrix},$$

ii. The Hessian is $\mathbf{A}$ and the gradient is $\mathbf{A}\mathbf{x} + \mathbf{d}$. **iii.** The eigenvalues

of the Hessian can be found by

$$|\mathbf{A} - \lambda\mathbf{I}| = 0 = \left| \begin{bmatrix} 34 - \lambda & \\ -12 - 12 & 16 - \lambda \end{bmatrix} \right| = \lambda^2 - 50\lambda + 400 = (\lambda - 10)(\lambda - 40)$$

So the eigenvalues are 10 and 40. The corresponding eigenvectors can be found using

$$\left[\mathbf{A} - \lambda\mathbf{I}\right]\mathbf{v} = \mathbf{0}$$

For $\lambda_1 = 10$ we have

$$\begin{bmatrix} 24 & -12 \\ -12 & 6 \end{bmatrix} \mathbf{v}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{v}_1 = \begin{bmatrix} -1 \\ -2 \end{bmatrix}$$

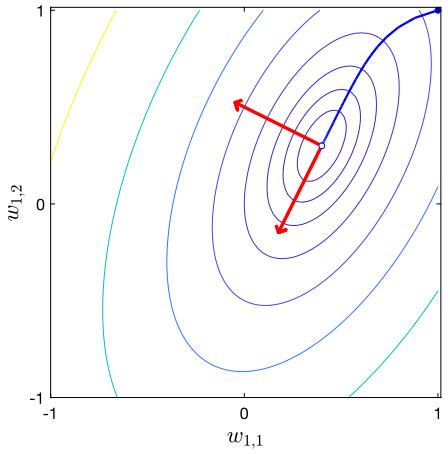
For $\lambda_2 = 40$ we have

$$\begin{bmatrix} -6 & -12 \\ -12 & -24 \end{bmatrix} \mathbf{v}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{v}_2 = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

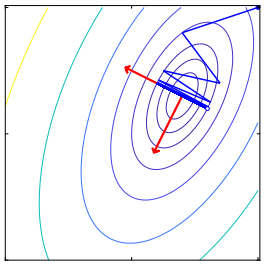
iv. The contours of $F(\mathbf{x})$ will be elliptical, and the major axis of each ellipse will be $\mathbf{v}_1$, since this is the eigenvector with the smallest eigenvalue (curvature). The center of the contours will be at the point where the gradient is equal to zero.

$$\mathbf{A}\mathbf{x} + \mathbf{d} = \mathbf{0} \Rightarrow \mathbf{x} = -\mathbf{A}^{-1}\mathbf{d} = \begin{bmatrix} 34 & -12 \\ -12 & 16 \end{bmatrix}^{-1} \begin{bmatrix} -10 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 0.3 \end{bmatrix}$$

The figure below shows the eigenvectors and the contours. The GD path will move down hill and orthogonal to the contour lines.

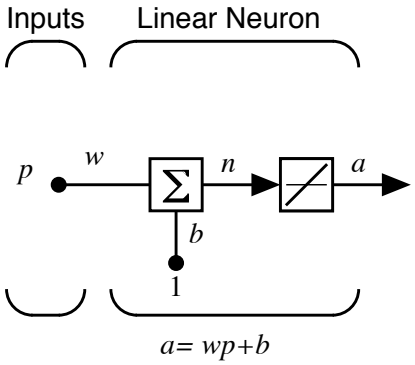


v. Since the largest eigenvalue of the Hessian is 40, the maximum stable learning rate will be $\alpha = 2/40 = 0.05$. With this learning rate, the GD trajectory will oscillate back and forth along the eigenvector that corresponds to the maximum eigenvalue, which would be $\mathbf{v}_2$. The figure in the margin shows what the trajectory would look like.



Exercises

E3.1 Consider the training data $\{p_1 = [-1], t_1 = [-1.5]\}$, $\{p_2 = [0], t_2 = [0.5]\}$, $\{p_3 = [1], t_3 = [2.5]\}$ and the network below.



i. Write out an expression for the sum squared error.

ii. Starting with initial weight and bias equal to zero, calculate the initial gradient of the sum squared error.

iii. Take one step of steepest descent with learning rate of 0.1.

iv. Does the error go down?

E3.2 Use the network and data from Exercise E3.1.

i. Find the gradient and Hessian of the sum squared error.

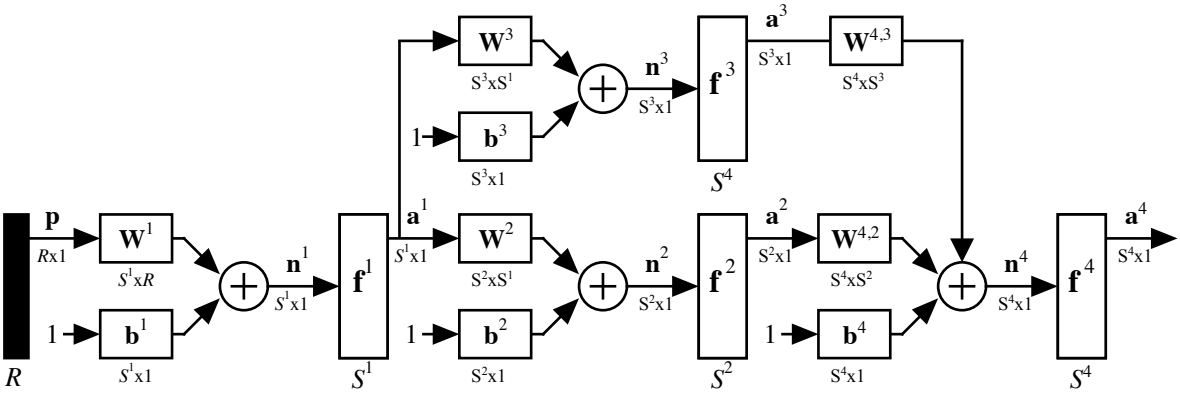
ii. Starting with initial weight and bias equal to zero, take a step of Newton's method – Eq. 3.28

iii. After a step of Newton's method, has the algorithm reached the minimum sum squared error? (Check that the gradient is zero.) Explain how Newton's method has used curvature to make a better first step than steepest descent.

E3.3 Consider again the network from Exercise E2.3. Use the weights and biases that you found there. Replace the hard limit transfer function with a linear transfer function. For input $\mathbf{p} = \begin{bmatrix} 0 & -1 \end{bmatrix}^T$ and target

$\mathbf{t} = [1]$, compute the initial gradient using backpropagation, and perform one iteration of steepest descent with a learning rate of 0.1.

E3.4 The network shown in the following figure is not a standard multilayer network, since the output of one layer does not feed into only one succeeding layer. However, it is a purely feedforward network, and the standard backpropagation techniques can be applied to compute the training gradients. (We will say more about computing gradients for arbitrarily connected layers in a future chapter.) Write out the equations needed to compute the sensitivities $\mathbf{s}^4, \mathbf{s}^3, \mathbf{s}^2$ and $\mathbf{s}^1$ for this network, using the concepts that were used to derive Eq. 3.49.



E3.5 The initialization of the backpropagation algorithm, Eq. 3.50, depends on the performance function and the last layer activation function. Find the specific initialization step, if the performance function is cross-entropy, and the last layer activation function is $softmax$.

E3.6 If the cross-entropy performance function is used, show the explicit form for the initialization of the backpropagation process (Eq. 3.50).