CHAPTER 2

Training Feed-Forward Neural Networks

The Fast-Food Problem

We're beginning to understand how we can tackle some interesting problems using deep learning, but one big question still remains: how exactly do we figure out what the parameter vectors (the weights for all of the connections in our neural network) should be? This is accomplished by a process commonly referred to as *training* (see Figure 2-1). During training, we show the neural net a large number of training examples and iteratively modify the weights to minimize the errors we make on the training examples. After enough examples, we expect that our neural network will be quite effective at solving the task it's been trained to do.

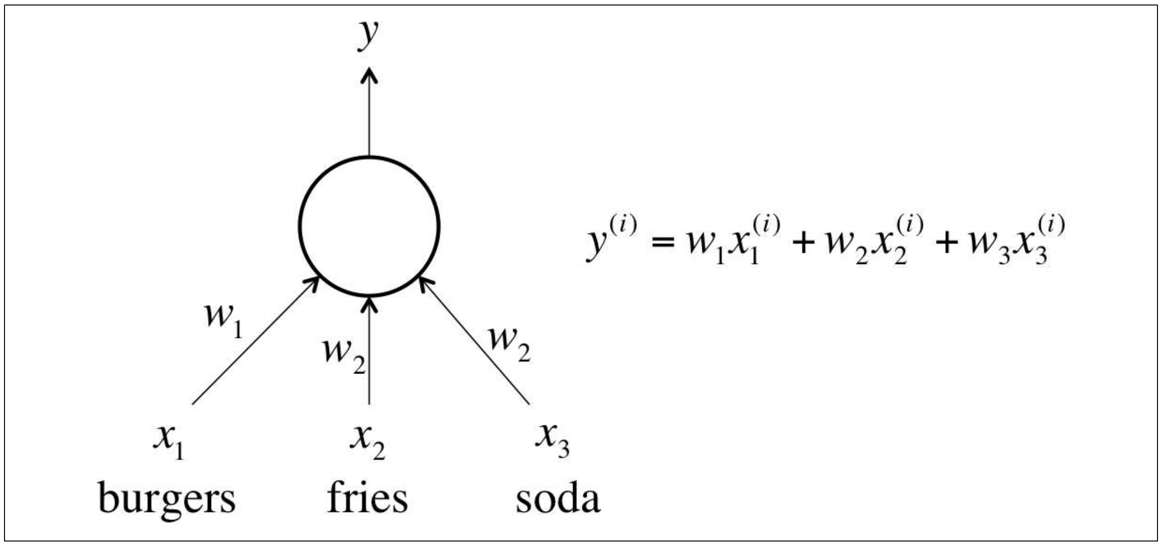

Figure 2-1. This is the neuron we want to train for the fast-food problem

Let’s continue with the example we mentioned in the previous chapter involving a linear neuron. As a brief review, every single day, we purchase a restaurant meal consisting of burgers, fries, and sodas. We buy some number of servings for each item. We want to be able to predict how much a meal is going to cost us, but the items don’t have price tags. The only thing the cashier will tell us is the total price of the meal. We want to train a single linear neuron to solve this problem. How do we do it?

One idea is to be intelligent about picking our training cases. For one meal we could buy only a single serving of burgers, for another we could only buy a single serving of fries, and then for our last meal we could buy a single serving of soda. In general, intelligently selecting training examples is a very good idea. There’s lots of research that shows that by engineering a clever training set, you can make your neural network a lot more effective. The issue with using this approach alone is that in real situations, it rarely ever gets you close to the solution. For example, there’s no clear analog of this strategy in image recognition. It’s just not a practical solution.

Instead, we try to motivate a solution that works well in general. Let’s say we have a large set of training examples. Then we can calculate what the neural network will output on the i^{th} training example using the simple formula in the diagram. We want to train the neuron so that we pick the optimal weights possible—the weights that minimize the errors we make on the training examples. In this case, let’s say we want to minimize the square error over all of the training examples that we encounter. More formally, if we know that $t^{(i)}$ is the true answer for the i^{th} training example and $y^{(i)}$ is the value computed by the neural network, we want to minimize the value of the error function E:

$$E = \frac{1}{2}\Sigma_i\left(t^{(i)} - y^{(i)}\right)^2$$

The squared error is zero when our model makes a perfectly correct prediction on every training example. Moreover, the closer E is to 0, the better our model is. As a result, our goal will be to select our parameter vector θ (the values for all the weights in our model) such that E is as close to 0 as possible.

Now at this point you might be wondering why we need to bother ourselves with error functions when we can treat this problem as a system of equations. After all, we have a bunch of unknowns (weights) and we have a set of equations (one for each training example). That would automatically give us an error of 0, assuming that we have a consistent set of training examples.

That’s a smart observation, but the insight unfortunately doesn’t generalize well. Remember that although we’re using a linear neuron here, linear neurons aren’t used very much in practice because they’re constrained in what they can learn. And the moment we start using nonlinear neurons like the sigmoidal, tanh, or ReLU neurons we talked about at the end of the previous chapter, we can no longer set up a system of equations! Clearly we need a better strategy to tackle the training process.

Gradient Descent

Let's visualize how we might minimize the squared error over all of the training examples by simplifying the problem. Let's say our linear neuron only has two inputs (and thus only two weights, w_1 and w_2). Then we can imagine a three-dimensional space where the horizontal dimensions correspond to the weights w_1 and w_2, and the vertical dimension corresponds to the value of the error function E. In this space, points in the horizontal plane correspond to different settings of the weights, and the height at those points corresponds to the incurred error. If we consider the errors we make over all possible weights, we get a surface in this three-dimensional space, in particular, a quadratic bowl as shown in Figure 2-2.

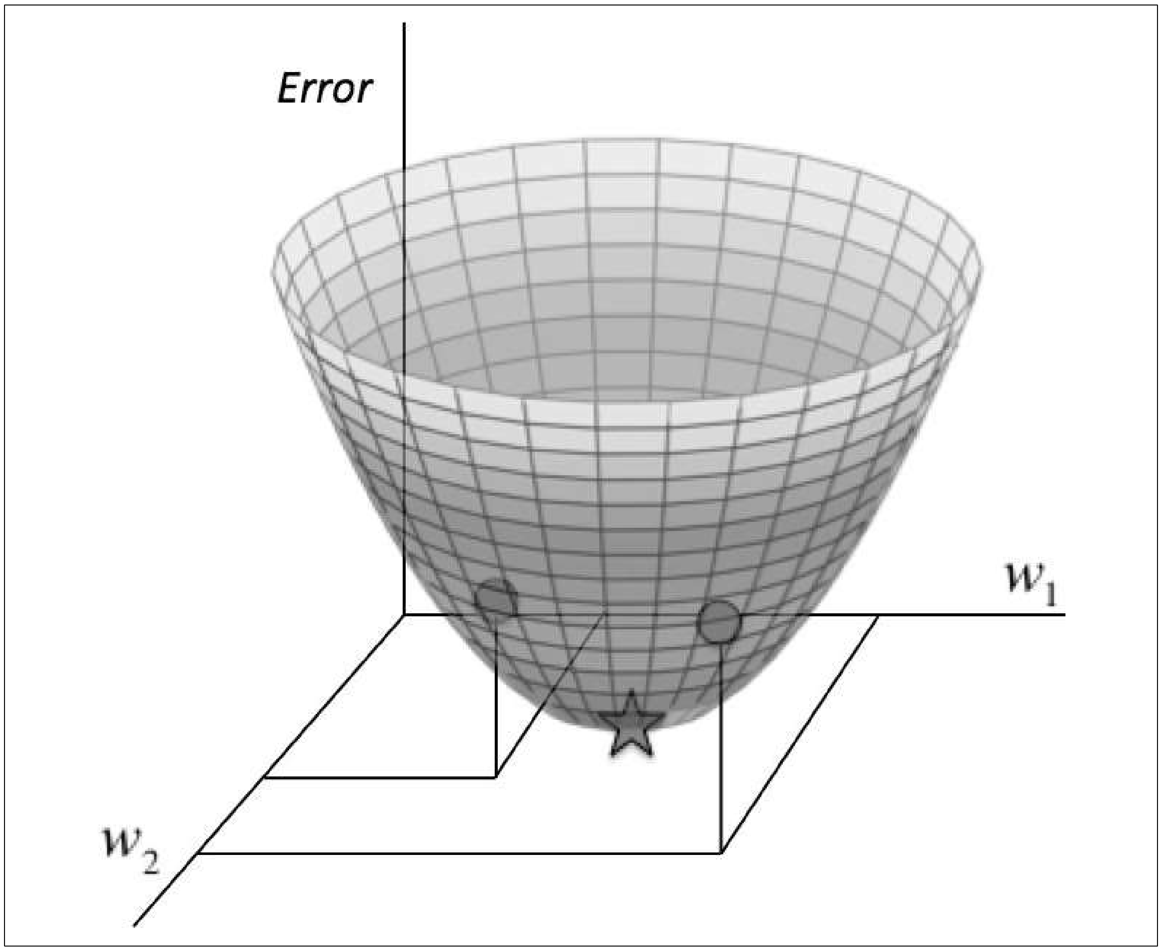

Figure 2-2. The quadratic error surface for a linear neuron

We can also conveniently visualize this surface as a set of elliptical contours, where the minimum error is at the center of the ellipses. In this setup, we are working in a two-dimensional plane where the dimensions correspond to the two weights. Contours correspond to settings of w_1 and w_2 that evaluate to the same value of E. The closer the contours are to each other, the steeper the slope. In fact, it turns out that the direction of the steepest descent is always perpendicular to the contours. This direction is expressed as a vector known as the *gradient*.

Now we can develop a high-level strategy for how to find the values of the weights that minimizes the error function. Suppose we randomly initialize the weights of our network so we find ourselves somewhere on the horizontal plane. By evaluating the gradient at our current position, we can find the direction of steepest descent, and we can take a step in that direction. Then we'll find ourselves at a new position that's closer to the minimum than we were before. We can reevaluate the direction of steepest descent by taking the gradient at this new position and taking a step in this new direction. It's easy to see that, as shown in Figure 2-3, following this strategy will eventually get us to the point of minimum error. This algorithm is known as *gradient descent*, and we'll use it to tackle the problem of training individual neurons and the more general challenge of training entire networks.[1]

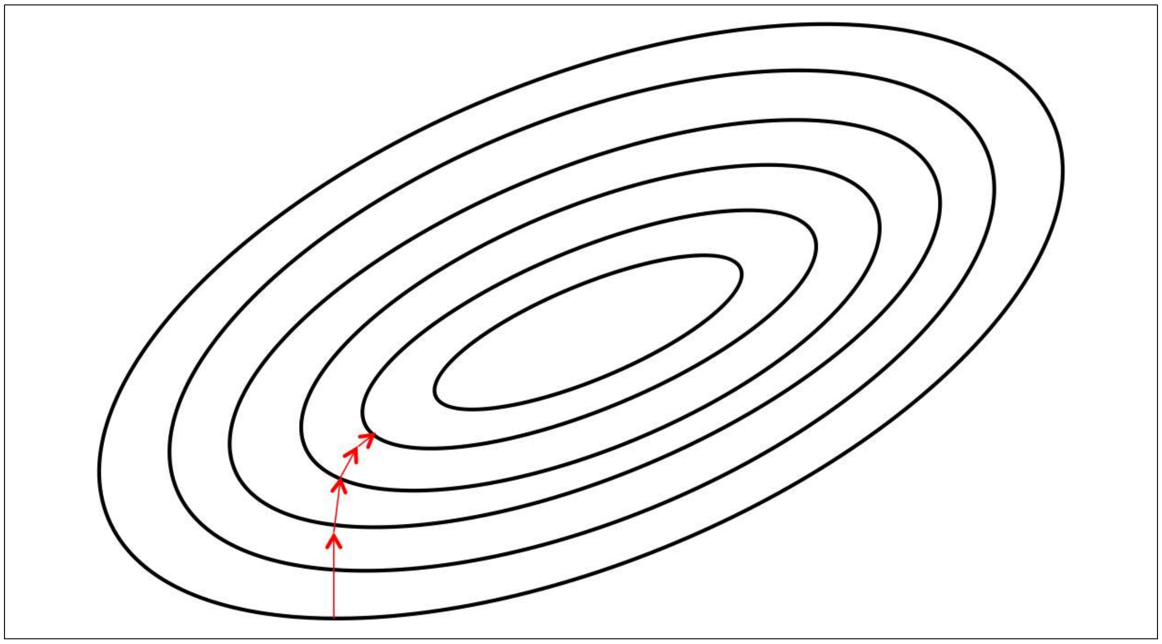

Figure 2-3. Visualizing the error surface as a set of contours

1 Rosenbloom, P. "The method of steepest descent." *Proceedings of Symposia in Applied Mathematics*. Vol. 6. 1956.

The Delta Rule and Learning Rates

Before we derive the exact algorithm for training our fast-food neuron, a quick note on *hyperparameters*. In addition to the weight parameters defined in our neural network, learning algorithms also require a couple of additional parameters to carry out the training process. One of these so-called hyperparameters is the *learning rate*.

In practice, at each step of moving perpendicular to the contour, we need to determine how far we want to walk before recalculating our new direction. This distance needs to depend on the steepness of the surface. Why? The closer we are to the minimum, the shorter we want to step forward. We know we are close to the minimum, because the surface is a lot flatter, so we can use the steepness as an indicator of how close we are to the minimum. However, if our error surface is rather mellow, training can potentially take a large amount of time. As a result, we often multiply the gradient by a factor ϵ, the learning rate. Picking the learning rate is a hard problem (Figure 2-4). As we just discussed, if we pick a learning rate that's too small, we risk taking too long during the training process. But if we pick a learning rate that's too big, we'll mostly likely start diverging away from the minimum. In Chapter 3, we'll learn about various optimization techniques that utilize adaptive learning rates to automate the process of selecting learning rates.

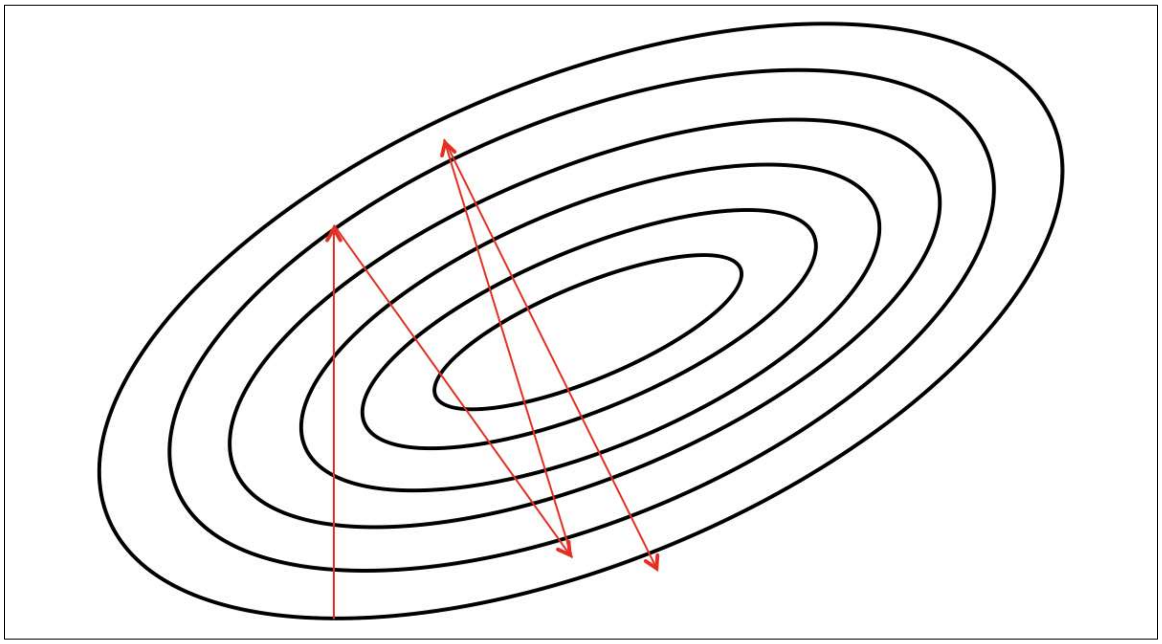

Figure 2-4. Convergence is difficult when our learning rate is too large

Now, we are finally ready to derive the *delta rule* for training our linear neuron. In order to calculate how to change each weight, we evaluate the gradient, which is essentially the partial derivative of the error function with respect to each of the weights. In other words, we want:

$$\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k}$$

$$= -\epsilon \frac{\partial}{\partial w_k}\left(\frac{1}{2}\Sigma_i \left(t^{(i)} - y^{(i)}\right)^2\right)$$

$$= \Sigma_i \epsilon\left(t^{(i)} - y^{(i)}\right)\frac{\partial y_i}{\partial w_k}$$

$$= \Sigma_i \epsilon x_k^{(i)}\left(t^{(i)} - y^{(i)}\right)$$

Applying this method of changing the weights at every iteration, we are finally able to utilize gradient descent.

Gradient Descent with Sigmoidal Neurons

In this section and the next, we will deal with training neurons and neural networks that utilize nonlinearities. We use the sigmoidal neuron as a model, and leave the derivations for other nonlinear neurons as an exercise for the reader. For simplicity, we assume that the neurons do not use a bias term, although our analysis easily extends to this case. We merely need to assume that the bias is a weight on an incoming connection whose input value is always one.

Let's recall the mechanism by which logistic neurons compute their output value from their inputs:

$$z = \Sigma_k w_k x_k$$

$$y = \frac{1}{1 + e^{-z}}$$

The neuron computes the weighted sum of its inputs, the logit z. It then feeds its logit into the input function to compute y, its final output. Fortunately for us, these functions have very nice derivatives, which makes learning easy! For learning, we want to compute the gradient of the error function with respect to the weights. To do so, we start by taking the derivative of the logit with respect to the inputs and the weights:

$$\frac{\partial z}{\partial w_k} = x_k$$

$$\frac{\partial z}{\partial x_k} = w_k$$

Also, quite surprisingly, the derivative of the output with respect to the logit is quite simple if you express it in terms of the output:

$$\frac{dy}{dz} = \frac{e^{-z}}{\left(1 + e^{-z}\right)^2}$$

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

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

$$= y(1-y)$$

We then use the chain rule to get the derivative of the output with respect to each weight:

$$\frac{\partial y}{\partial w_k} = \frac{dy}{dz}\frac{\partial z}{\partial w_k} = x_k y(1-y)$$

Putting all of this together, we can now compute the derivative of the error function with respect to each weight:

$$\frac{\partial E}{\partial w_k} = \sum_i \frac{\partial E}{\partial y^{(i)}}\frac{\partial y^{(i)}}{\partial w_k} = -\sum_i x_k^{(i)} y^{(i)}\left(1-y^{(i)}\right)\left(t^{(i)}-y^{(i)}\right)$$

Thus, the final rule for modifying the weights becomes:

$$\Delta w_k = \sum_i \epsilon x_k^{(i)} y^{(i)}\left(1-y^{(i)}\right)\left(t^{(i)}-y^{(i)}\right)$$

As you may notice, the new modification rule is just like the delta rule, except with extra multiplicative terms included to account for the logistic component of the sigmoidal neuron.

The Backpropagation Algorithm

Now we're finally ready to tackle the problem of training multilayer neural networks (instead of just single neurons). To accomplish this task, we'll use an approach known as *backpropagation*, pioneered by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams in 1986.[2] So what's the idea behind backpropagation? We don't know what the hidden units ought to be doing, but what we can do is compute how fast the error changes as we change a hidden activity. From there, we can figure out how fast the error changes when we change the weight of an individual connection. Essentially, we'll be trying to find the path of steepest descent! The only catch is that we're going to be working in an extremely high-dimensional space. We start by calculating the error derivatives with respect to a single training example.

Each hidden unit can affect many output units. Thus, we'll have to combine many separate effects on the error in an informative way. Our strategy will be one of dynamic programming. Once we have the error derivatives for one layer of hidden

2 Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Cognitive Modeling* 5.3 (1988): 1.

units, we'll use them to compute the error derivatives for the activities of the layer below. And once we find the error derivatives for the activities of the hidden units, it's quite easy to get the error derivatives for the weights leading into a hidden unit. We'll redefine some notation for ease of discussion and refer to the Figure 2-5.

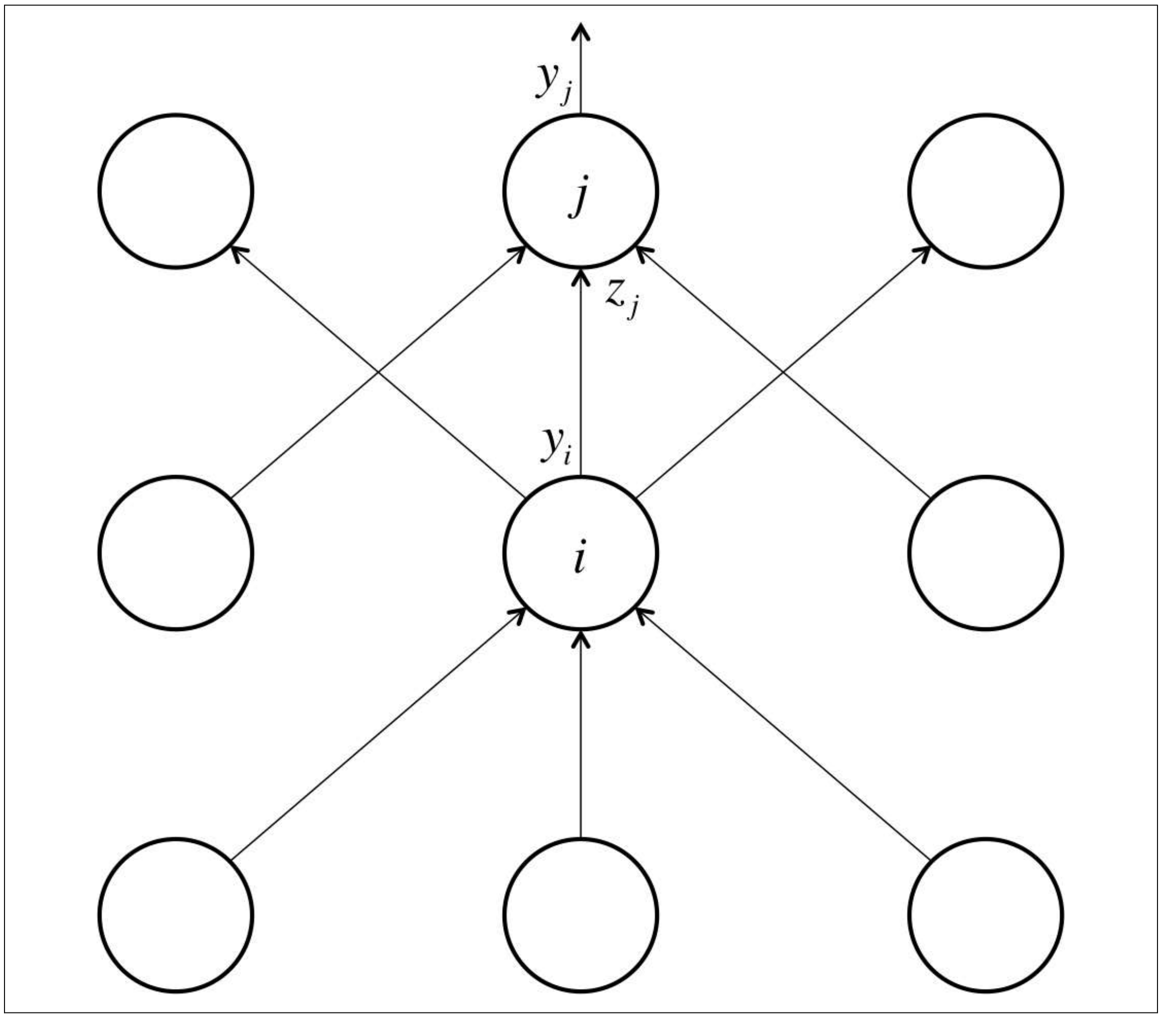

Figure 2-5. Reference diagram for the derivation of the backpropagation algorithm

The subscript we use will refer to the layer of the neuron. The symbol y will refer to the activity of a neuron, as usual. Similarly, the symbol z will refer to the logit of the neuron. We start by taking a look at the base case of the dynamic programming problem. Specifically, we calculate the error function derivatives at the output layer:

$$E = \frac{1}{2}\Sigma_{j \in output}\left(t_j - y_j\right)^2 \Rightarrow \frac{\partial E}{\partial y_j} = -\left(t_j - y_j\right)$$

Now we tackle the inductive step. Let's presume we have the error derivatives for layer j. We now aim to calculate the error derivatives for the layer below it, layer i. To do so, we must accumulate information about how the output of a neuron in layer i affects the logits of every neuron in layer j. This can be done as follows, using the fact that the partial derivative of the logit with respect to the incoming output data from the layer beneath is merely the weight of the connection w_{ij}:

$$\frac{\partial E}{\partial y_i} = \Sigma_j \frac{\partial E}{\partial z_j}\frac{dz_j}{dy_i} = \Sigma_j w_{ij}\frac{\partial E}{\partial z_j}$$

Furthermore, we observe the following:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{dy_j}{dz_j} = y_j\left(1 - y_j\right)\frac{\partial E}{\partial y_j}$$

Combining these two together, we can finally express the error derivatives of layer *i* in terms of the error derivatives of layer *j*:

$$\frac{\partial E}{\partial y_i} = \Sigma_j w_{ij} y_j\left(1 - y_j\right)\frac{\partial E}{\partial y_j}$$

Then once we've gone through the whole dynamic programming routine, having filled up the table appropriately with all of our partial derivatives (of the error function with respect to the hidden unit activities), we can then determine how the error changes with respect to the weights. This gives us how to modify the weights after each training example:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i y_j\left(1 - y_j\right)\frac{\partial E}{\partial y_j}$$

Finally, to complete the algorithm, just as before, we merely sum up the partial derivatives over all the training examples in our dataset. This gives us the following modification formula:

$$\Delta w_{ij} = -\Sigma_{k \in dataset}\, \epsilon y_i^{(k)} y_j^{(k)}\left(1 - y_j^{(k)}\right)\frac{\partial E^{(k)}}{\partial y_j^{(k)}}$$

This completes our description of the backpropagation algorithm!

Stochastic and Minibatch Gradient Descent

In the algorithms we've described in "The Backpropagation Algorithm" on page 23, we've been using a version of gradient descent known as *batch gradient descent*. The idea behind batch gradient descent is that we use our entire dataset to compute the error surface and then follow the gradient to take the path of steepest descent. For a simple quadratic error surface, this works quite well. But in most cases, our error surface may be a lot more complicated. Let's consider the scenario in Figure 2-6 for illustration.

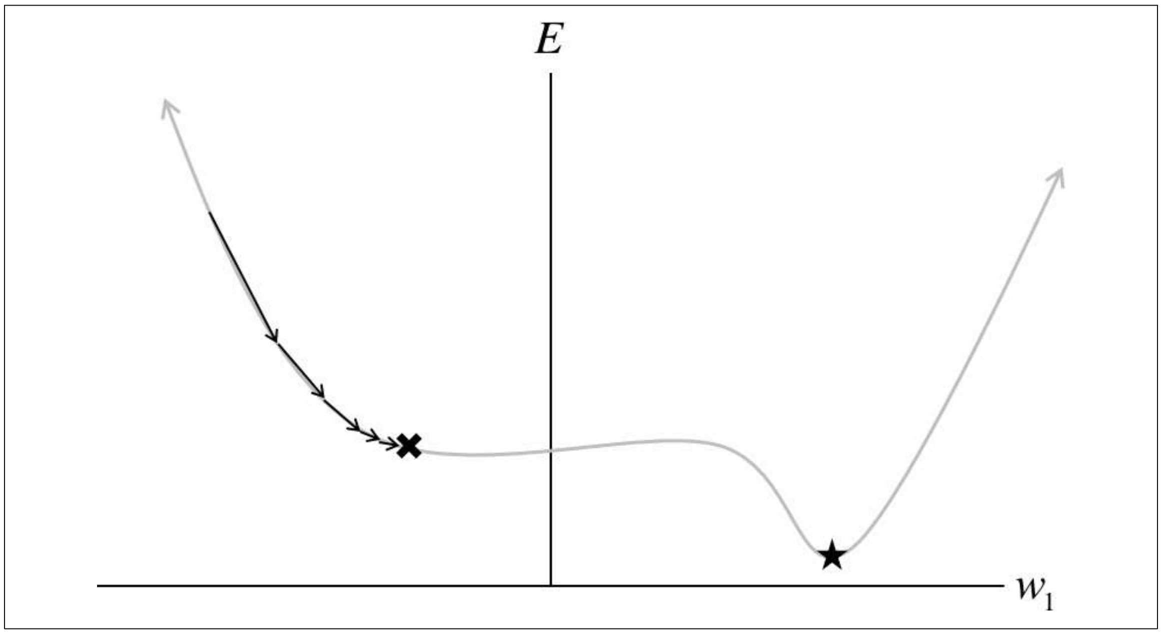

Figure 2-6. Batch gradient descent is sensitive to saddle points, which can lead to premature convergence

We only have a single weight, and we use random initialization and batch gradient descent to find its optimal setting. The error surface, however, has a flat region (also known as saddle point in high-dimensional spaces), and if we get unlucky, we might find ourselves getting stuck while performing gradient descent.

Another potential approach is *stochastic gradient descent* (SGD), where at each iteration, our error surface is estimated only with respect to a single example. This approach is illustrated by Figure 2-7, where instead of a single static error surface, our error surface is dynamic. As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions.

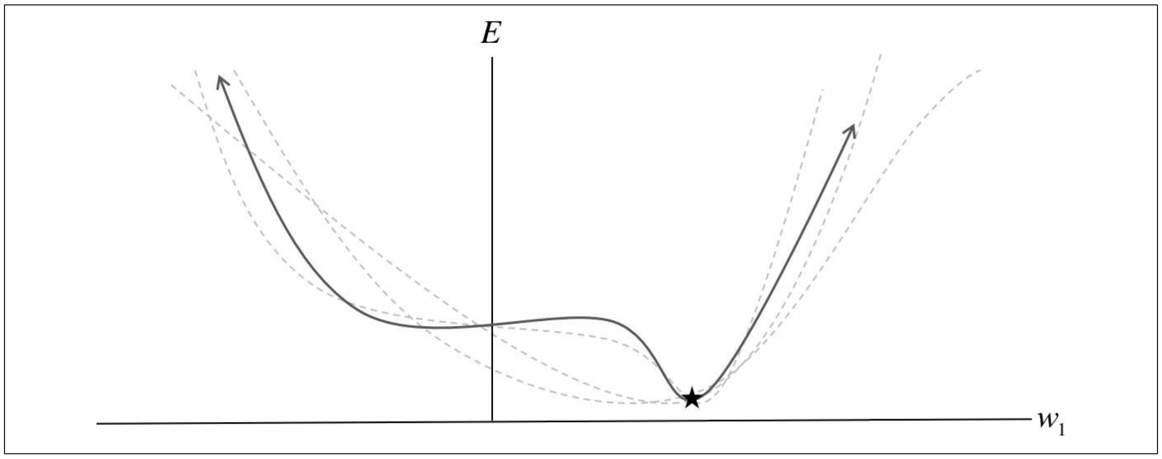

Figure 2-7. The stochastic error surface fluctuates with respect to the batch error surface, enabling saddle point avoidance

The major pitfall of stochastic gradient descent, however, is that looking at the error incurred one example at a time may not be a good enough approximation of the error surface. This, in turn, could potentially make gradient descent take a significant amount of time. One way to combat this problem is using *mini-batch gradient descent*. In mini-batch gradient descent, at every iteration, we compute the error surface with respect to some subset of the total dataset (instead of just a single example). This subset is called a *minibatch*, and in addition to the learning rate, minibatch size is another hyperparameter. Minibatches strike a balance between the efficiency of batch gradient descent and the local-minima avoidance afforded by stochastic gradient descent. In the context of backpropagation, our weight update step becomes:

$$\Delta w_{ij} = -\Sigma_{k \in minibatch} \epsilon y_i^{(k)} y_j^{(k)} \left(1 - y_j^{(k)}\right) \frac{\partial E^{(k)}}{\partial y_j^{(k)}}$$

This is identical to what we derived in the previous section, but instead of summing over all the examples in the dataset, we sum over the examples in the current minibatch.

Test Sets, Validation Sets, and Overfitting

One of the major issues with artificial neural networks is that the models are quite complicated. For example, let's consider a neural network that's pulling data from an image from the MNIST database (28 x 28 pixels), feeds into two hidden layers with 30 neurons, and finally reaches a softmax layer of 10 neurons. The total number of parameters in the network is nearly 25,000. This can be quite problematic, and to understand why, let's consider a new toy example, illustrated in Figure 2-8.

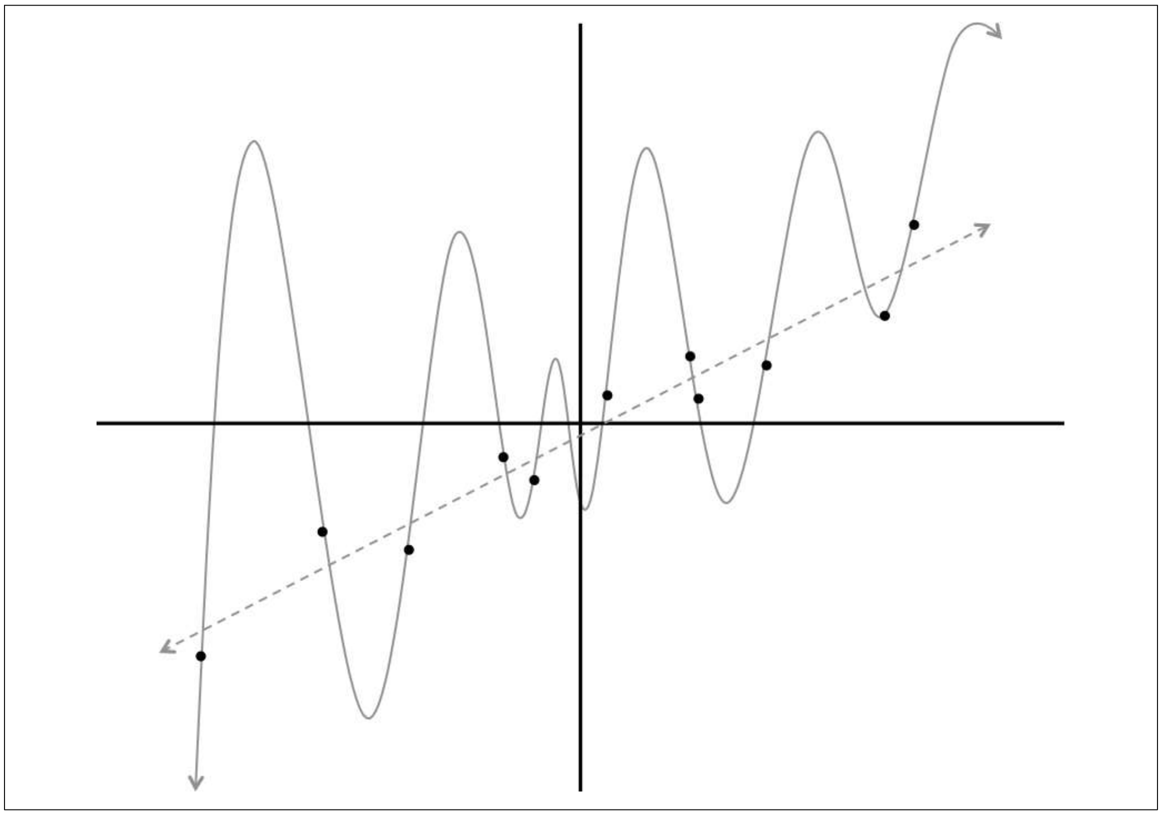

Figure 2-8. Two potential models that might describe our dataset: a linear model versus a degree 12 polynomial

We are given a bunch of data points on a flat plane, and our goal is to find a curve that best describes this dataset (i.e., will allow us to predict the y-coordinate of a new point given its x-coordinate). Using the data, we train two different models: a linear model and a degree 12 polynomial. Which curve should we trust? The line which gets almost no training example correctly? Or the complicated curve that hits every single point in the dataset? At this point we might trust the linear fit because it seems much less contrived. But just to be sure, let's add more data to our dataset! The result is shown in Figure 2-9.

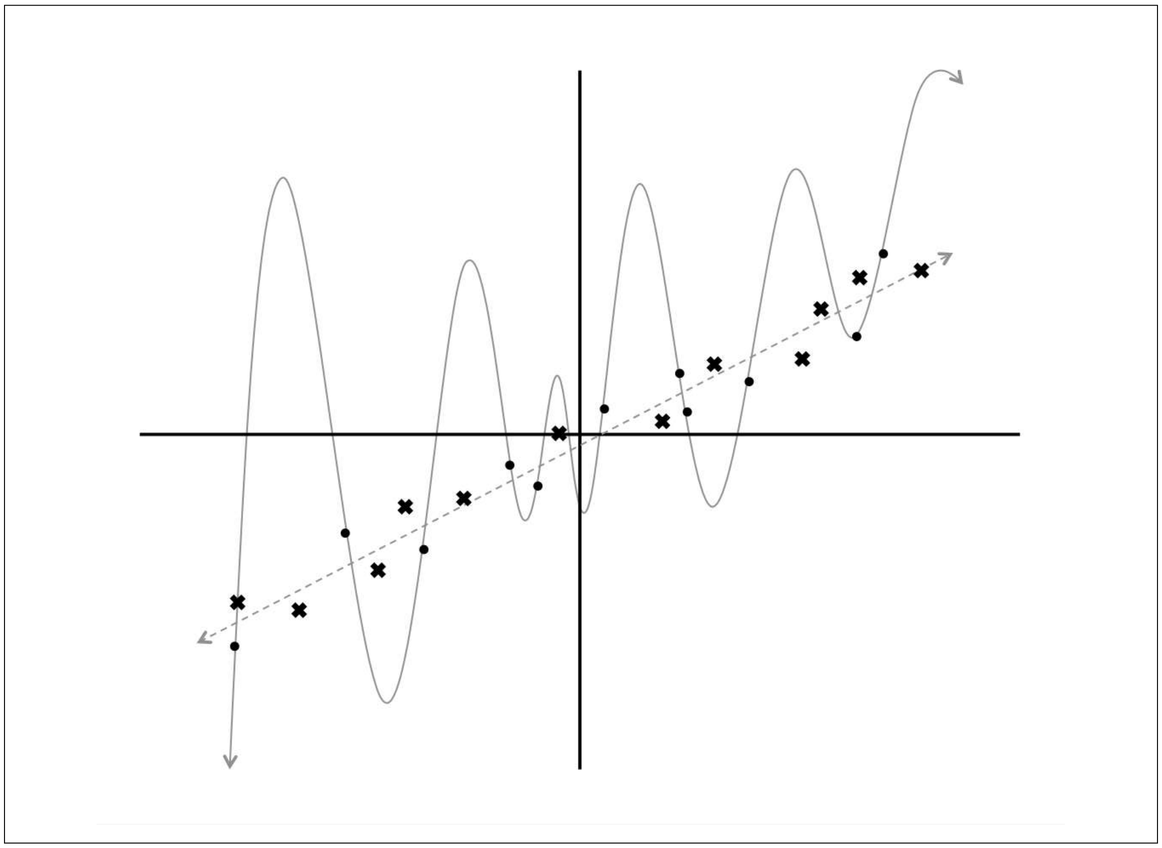

Figure 2-9. Evaluating our model on new data indicates that the linear fit is a much better model than the degree 12 polynomial

Now the verdict is clear: the linear model is not only better subjectively but also quantitatively (measured using the squared error metric). But this leads to a very interesting point about training and evaluating machine learning models. By building a very complex model, it's quite easy to perfectly fit our training dataset because we give our model enough degrees of freedom to contort itself to fit the observations in the training set. But when we evaluate such a complex model on new data, it performs very poorly. In other words, the model does not *generalize* well. This is a phenomenon called *overfitting*, and it is one of the biggest challenges that a machine learning engineer must combat. This becomes an even more significant issue in deep learning, where our neural networks have large numbers of layers containing many neurons. The number of connections in these models is astronomical, reaching the millions. As a result, overfitting is commonplace.

Let's see how this looks in the context of a neural network. Let's say we have a neural network with two inputs, a softmax output of size two, and a hidden layer with 3, 6, or 20 neurons. We train these networks using mini-batch gradient descent (batch size 10), and the results, visualized using ConvNetJS, are shown in Figure 2-10.[3]

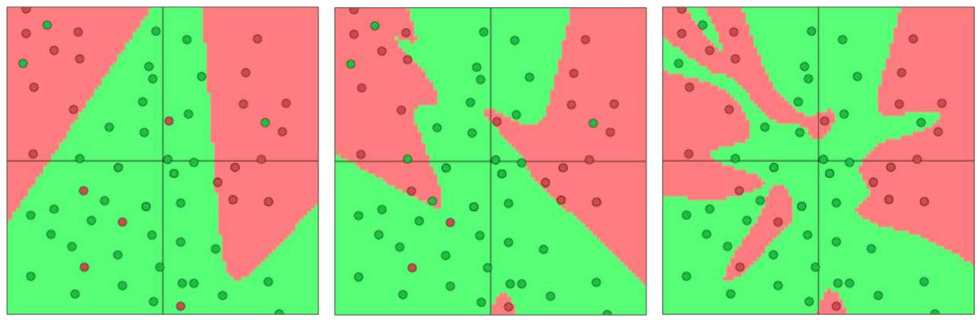

Figure 2-10. A visualization of neural networks with 3, 6, and 20 neurons (in that order) in their hidden layer

It's already quite apparent from these images that as the number of connections in our network increases, so does our propensity to overfit to the data. We can similarly see the phenomenon of overfitting as we make our neural networks deep. These results are shown in Figure 2-11, where we use networks that have one, two, or four hidden layers of three neurons each.

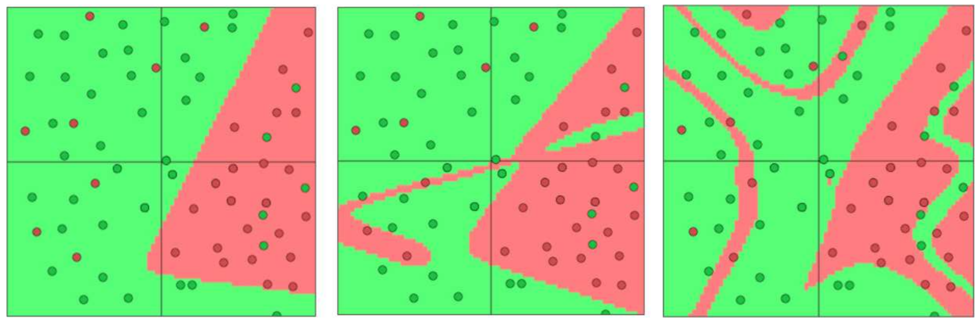

Figure 2-11. A visualization of neural networks with one, two, and four hidden layers (in that order) of three neurons each

This leads to three major observations. First, the machine learning engineer is always working with a direct trade-off between overfitting and model complexity. If the model isn't complex enough, it may not be powerful enough to capture all of the useful information necessary to solve a problem. However, if our model is very complex (especially if we have a limited amount of data at our disposal), we run the risk of overfitting. Deep learning takes the approach of solving very complex problems with complex models and taking additional countermeasures to prevent overfitting. We'll see a lot of these measures in this chapter as well as in later chapters.

3 *http://stanford.io/2pOdNhy*

Second, it is very misleading to evaluate a model using the data we used to train it. Using the example in Figure 2-8, this would falsely suggest that the degree 12 polynomial model is preferable to a linear fit. As a result, we almost never train our model on the entire dataset. Instead, as shown in Figure 2-12, we split up our data into a *training set* and a *test set*.

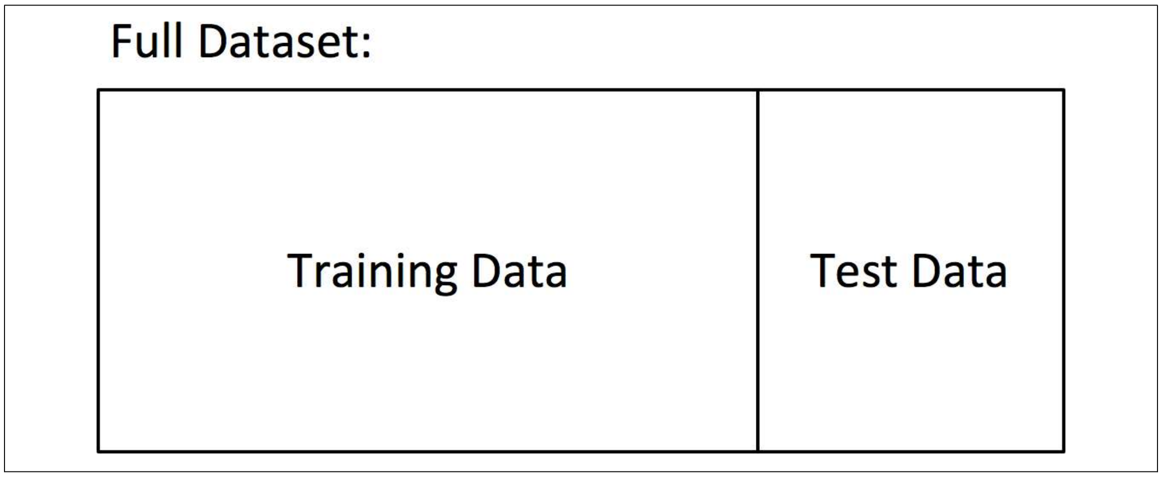

Figure 2-12. We often split our data into nonoverlapping training and test sets in order to fairly evaluate our model

This enables us to make a fair evaluation of our model by directly measuring how well it generalizes on new data it has not yet seen. In the real world, large datasets are hard to come by, so it might seem like a waste to not use all of the data at our disposal during the training process. Consequently, it may be very tempting to reuse training data for testing or cut corners while compiling test data. Be forewarned: if the test set isn't well constructed, we won't be able draw any meaningful conclusions about our model.

Third, it's quite likely that while we're training our data, there's a point in time where instead of learning useful features, we start overfitting to the training set. To avoid that, we want to be able to stop the training process as soon as we start overfitting, to prevent poor generalization. To do this, we divide our training process into *epochs*. An epoch is a single iteration over the entire training set. In other words, if we have a training set of size d and we are doing mini-batch gradient descent with batch size b, then an epoch would be equivalent to $\frac{d}{b}$ model updates. At the end of each epoch, we want to measure how well our model is generalizing. To do this, we use an additional *validation set*, which is shown in Figure 2-13. At the end of an epoch, the validation set will tell us how the model does on data it has yet to see. If the accuracy on the training set continues to increase while the accuracy on the validation set stays the same (or decreases), it's a good sign that it's time to stop training because we're overfitting.

The validation set is also helpful as a proxy measure of accuracy during the process of *hyperparameter optimization*. We've covered several hyperparameters so far in our discussion (learning rate, minibatch size, etc.), but we have yet to develop a framework for how to find the optimal values for these hyperparameters. One potential way to find the optimal setting of hyperparameters is by applying a *grid search,* where we pick a value for each hyperparameter from a finite set of options (e.g., $\epsilon \in \{0.001, 0.01, 0.1\}$, batch size $\in \{16, 64, 128\}, ...$), and train the model with every possible permutation of hyperparameter choices. We elect the combination of hyperparameters with the best performance on the validation set, and report the accuracy of the model trained with best combination on the test set.[4]

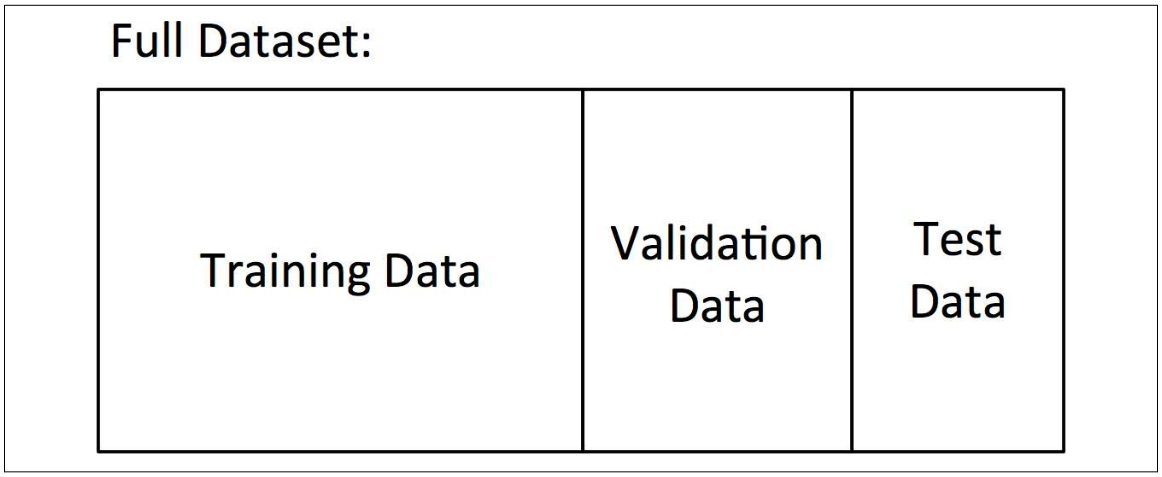

Figure 2-13. In deep learning we often include a validation set to prevent overfitting during the training process

With this in mind, before we jump into describing the various ways to directly combat overfitting, let's outline the workflow we use when building and training deep learning models. The workflow is described in detail in Figure 2-14. It is a tad intricate, but it's critical to understand the pipeline in order to ensure that we're properly training our neural networks.

First we define our problem rigorously. This involves determining our inputs, the potential outputs, and the vectorized representations of both. For instance, let's say our goal was to train a deep learning model to identify cancer. Our input would be an RBG image, which can be represented as a vector of pixel values. Our output would be a probability distribution over three mutually exclusive possibilities: 1) normal, 2) benign tumor (a cancer that has yet to metastasize), or 3) malignant tumor (a cancer that has already metastasized to other organs).

4 Nelder, John A., and Roger Mead. "A simplex method for function minimization." *The Computer Journal* 7.4 (1965): 308-313.

After we define our problem, we need to build a neural network architecture to solve it. Our input layer would have to be of appropriate size to accept the raw data from the image, and our output layer would have to be a softmax of size 3. We will also have to define the internal architecture of the network (number of hidden layers, the connectivities, etc.). We'll further discuss the architecture of image recognition models when we talk about convolutional neural networks in Chapter 4. At this point, we also want to collect a significant amount of data for training or modeling. This data would probably be in the form of uniformly sized pathological images that have been labeled by a medical expert. We shuffle and divide this data up into separate training, validation, and test sets.

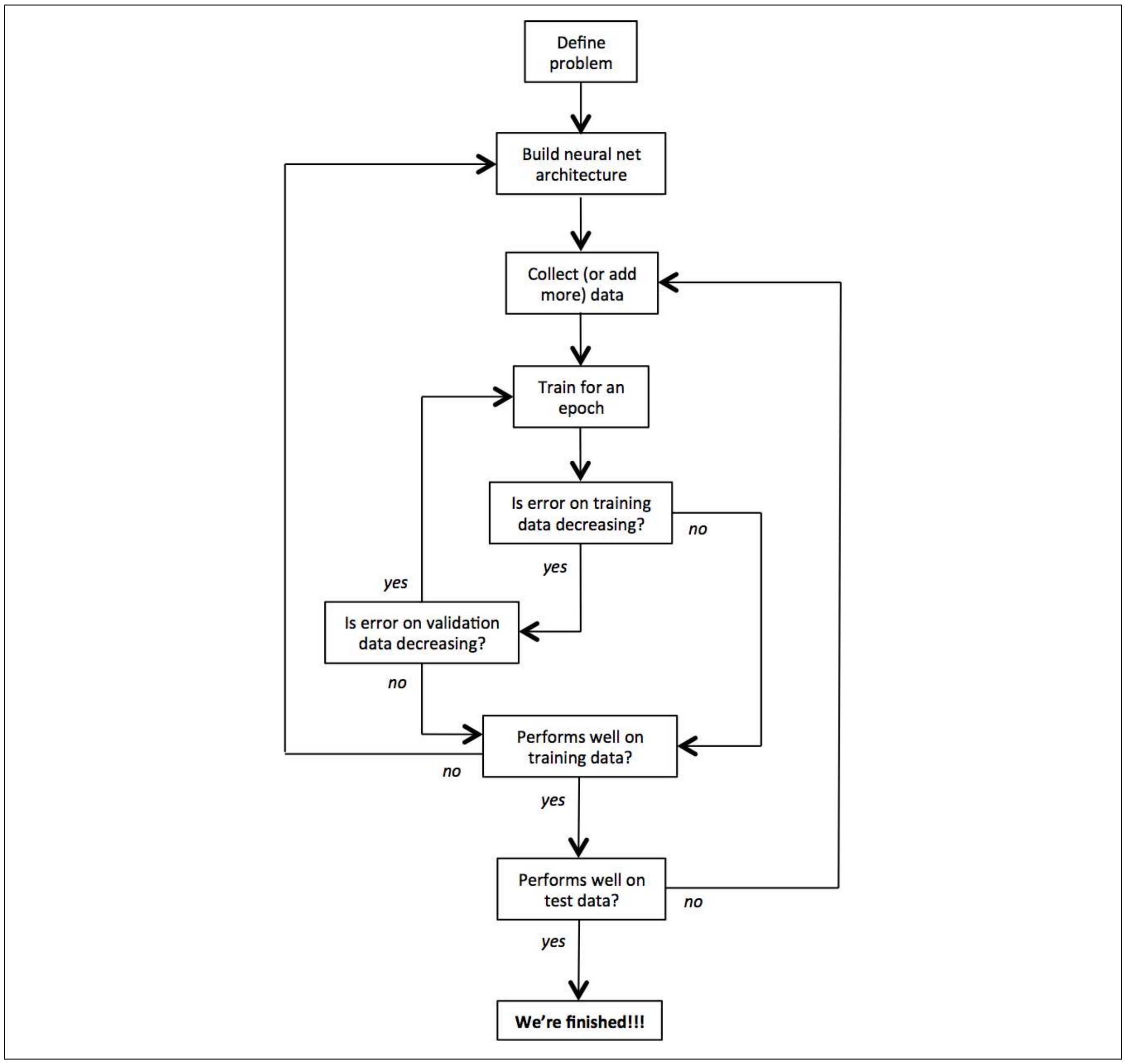
Define problem
Build neural net architecture
Collect (or add more) data
Train for an epoch
Is error on training data decreasing?
no
yes
yes
Is error on validation data decreasing?
no
Performs well on training data?
no
yes
Performs well on test data?
no
yes
We're finished!!!

Figure 2-14. Detailed workflow for training and evaluating a deep learning model

Finally, we're ready to begin gradient descent. We train the model on our training set for an epoch at a time. At the end of each epoch, we ensure that our error on the training set and validation set is decreasing. When one of these stops to improve, we terminate and make sure we're happy with the model's performance on the test data. If we're unsatisfied, we need to rethink our architecture or reconsider whether the data we collect has the information required to make the prediction we're interested in making. If our training set error stopped improving, we probably need to do a better job of capturing the important features in our data. If our validation set error stopped improving, we probably need to take measures to prevent overfitting.

If, however, we are happy with the performance of our model on the training data, then we can measure its performance on the test data, which the model has never seen before this point. If it is unsatisfactory, we need more data in our dataset because the test set seems to consist of example types that weren't well represented in the training set. Otherwise, we are finished!

Preventing Overfitting in Deep Neural Networks

There are several techniques that have been proposed to prevent overfitting during the training process. In this section, we'll discuss these techniques in detail.

One method of combatting overfitting is called *regularization*. Regularization modifies the objective function that we minimize by adding additional terms that penalize large weights. In other words, we change the objective function so that it becomes $Error + \lambda f(\theta)$, where $f(\theta)$ grows larger as the components of θ grow larger, and λ is the regularization strength (another hyperparameter). The value we choose for λ determines how much we want to protect against overfitting. A $\lambda = 0$ implies that we do not take any measures against the possibility of overfitting. If λ is too large, then our model will prioritize keeping θ as small as possible over trying to find the parameter values that perform well on our training set. As a result, choosing λ is a very important task and can require some trial and error.

The most common type of regularization in machine learning is *L2 regularization*.[5] It can be implemented by augmenting the error function with the squared magnitude of all weights in the neural network. In other words, for every weight w in the neural network, we add $\frac{1}{2}\lambda w^2$ to the error function. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. This has the appealing property of encouraging the network to use all of its inputs a little rather than using only some of its inputs a lot. Of particular note is that during the gradient descent update, using the L2 regularization ultimately means that every weight is decayed linearly to zero. Because of this phenomenon, L2 regularization is also commonly referred to as *weight decay*.

5 Tikhonov, Andrei Nikolaevich, and Vladlen Borisovich Glasko. "Use of the regularization method in nonlinear problems." *USSR Computational Mathematics and Mathematical Physics* 5.3 (1965): 93-107.

We can visualize the effects of L2 regularization using ConvNetJS. Similar to Figures 2-10 and 2-11, we use a neural network with 2 inputs, a softmax output of size 2, and a hidden layer with 20 neurons. We train the networks using mini-batch gradient descent (batch size 10) and regularization strengths of 0.01, 0.1, and 1. The results can be seen in Figure 2-15.

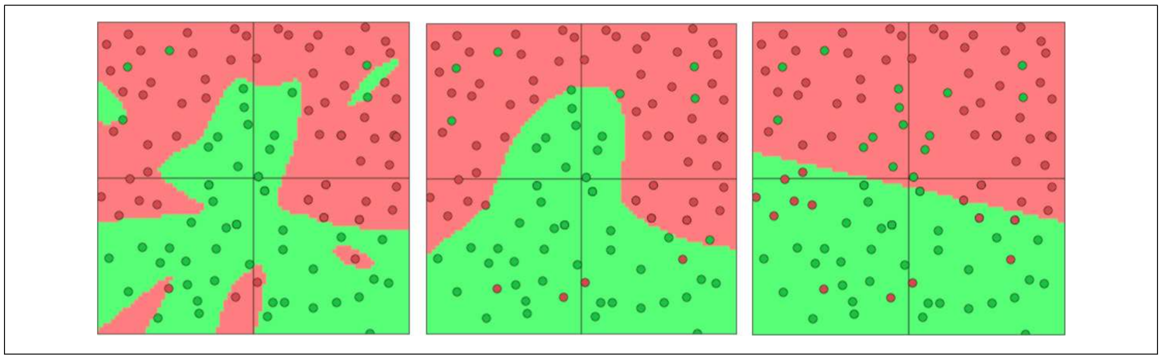

Figure 2-15. A visualization of neural networks trained with regularization strengths of 0.01, 0.1, and 1 (in that order)

Another common type of regularization is *L1 regularization*. Here, we add the term $\lambda|w|$ for every weight w in the neural network. The L1 regularization has the intriguing property that it leads the weight vectors to become sparse during optimization (i.e., very close to exactly zero). In other words, neurons with L1 regularization end up using only a small subset of their most important inputs and become quite resistant to noise in the inputs. In comparison, weight vectors from L2 regularization are usually diffuse, small numbers. L1 regularization is very useful when you want to understand exactly which features are contributing to a decision. If this level of feature analysis isn't necessary, we prefer to use L2 regularization because it empirically performs better.

Max norm constraints have a similar goal of attempting to restrict θ from becoming too large, but they do this more directly.[6] Max norm constraints enforce an absolute upper bound on the magnitude of the incoming weight vector for every neuron and use projected gradient descent to enforce the constraint. In other words, any time a gradient descent step moves the incoming weight vector such that $||w||_2 > c$, we project the vector back onto the ball (centered at the origin) with radius c. Typical values of c are 3 and 4. One of the nice properties is that the parameter vector cannot grow out of control (even if the learning rates are too high) because the updates to the weights are always bounded.

6 Srebro, Nathan, Jason DM Rennie, and Tommi S. Jaakkola. "Maximum-Margin Matrix Factorization." *NIPS*, Vol. 17, 2004.

Dropout is a very different kind of method for preventing overfitting that has become one of the most favored methods of preventing overfitting in deep neural networks. [7] While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. Intuitively, this forces the network to be accurate even in the absence of certain information. It prevents the network from becoming too dependent on any one (or any small combination) of neurons. Expressed more mathematically, it prevents overfitting by providing a way of approximately combining exponentially many different neural network architectures efficiently. The process of dropout is expressed pictorially in Figure 2-16.

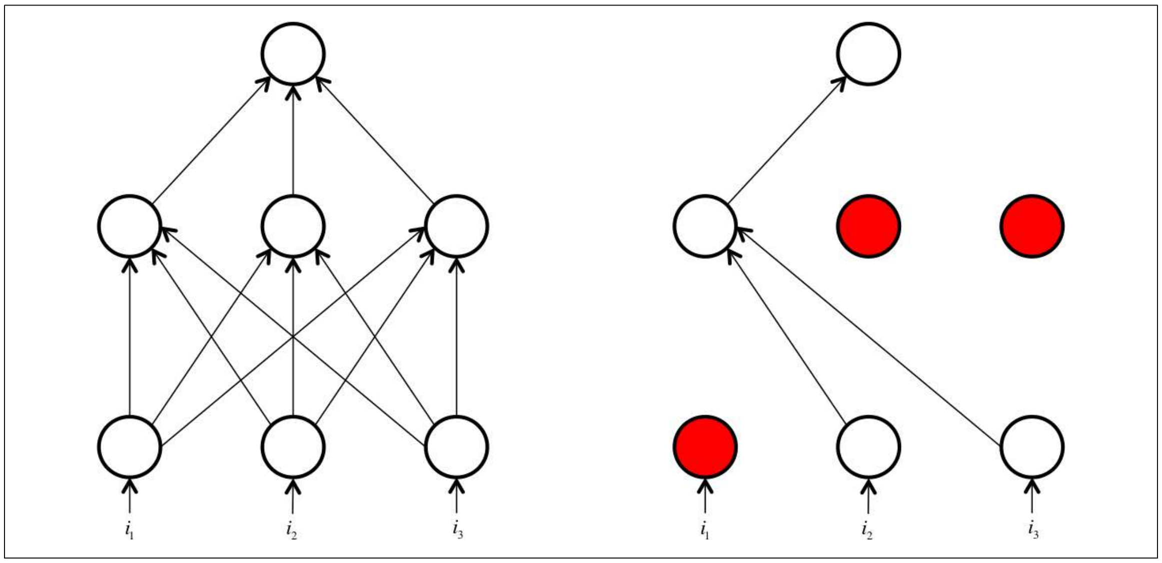

Figure 2-16. Dropout sets each neuron in the network as inactive with some random probability during each minibatch of training

Dropout is pretty intuitive to understand, but there are some important intricacies to consider. First, we'd like the outputs of neurons during test time to be equivalent to their expected outputs at training time. We could fix this naïvely by scaling the output at test time. For example, if $p = 0.5$, neurons must halve their outputs at test time in order to have the same (expected) output they would have during training. This is easy to see because a neuron's output is set to 0 with probability $1 - p$. This means that if a neuron's output prior to dropout was x, then after dropout, the expected output would be $E[\text{output}] = px + (1 - p) \cdot 0 = px$. This naïve implementation of dropout is undesirable, however, because it requires scaling of neuron outputs at test time. Test-time performance is extremely critical to model evaluation, so it's always preferable to use *inverted dropout*, where the scaling occurs at training time instead of at test

7 Srivastava, Nitish, et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

time. In inverted dropout, any neuron whose activation hasn't been silenced has its output divided by p before the value is propagated to the next layer. With this fix, $E[\text{output}] = p \cdot \frac{x}{p} + (1 - p) \cdot 0 = x$, and we can avoid arbitrarily scaling neuronal output at test time.

Summary

In this chapter, we've learned all of the basics involved in training feed-forward neural networks. We've talked about gradient descent, the backpropagation algorithm, as well as various methods we can use to prevent overfitting. In the next chapter, we'll put these lessons into practice when we use the TensorFlow library to efficiently implement our first neural networks. Then in Chapter 4, we'll return to the problem of optimizing objective functions for training neural networks and design algorithms to significantly improve performance. These improvements will enable us to process much more data, which means we'll be able to build more comprehensive models.