Chapter 14

Architecting Neural Networks

- Hyper-parameters
- Learning Rate & Momentum
- Hidden Structure
- Activation Functions

Hyper-parameters, as mentioned in previous chapters, are the numerous settings for models such as neural networks. Activation functions, hidden neuron counts, layer structure, convolution, max-pooling and dropout are all examples of neural network hyper-parameters. Finding the optimal set of hyper-parameters can seem a daunting task, and, indeed, it is one of the most time-consuming tasks for the AI programmer. However, do not fear, we will provide you with a summary of the current research on neural network architecture in this chapter. We will also show you how to conduct experiments to help determine the optimal architecture for your own networks.

We will make architectural recommendations in two ways. First, we will report on recommendations from scientific literature in the field of AI. These recommendations will include citations so that you can examine the original paper. However, we will strive to present the key concept of the article in an approachable manner. The second way will be through experimentation. We will run several competing architectures and report the results.

You need to remember that a few hard and fast rules do not dictate the optimal architecture for every project. Every data set is different, and, as a result, the optimal neural network for every data set is also different. Thus, you must always perform some experimentation to determine a good architecture for your network.

14.1 Evaluating Neural Networks

Neural networks start with random weights. Additionally, some training algorithms use random values as well. All considered, we're dealing with quite a bit of randomness in order to make comparisons. Random number seeds are a common solution to this issue; however, a constant seed does not provide an equal comparison, given that we are evaluating neural networks with different neuron counts.

Let's compare a neural network with 32 connections against another network with 64 connections. While the seed guarantees that the first 32 connections retain the same value, there are now 32 additional connections that will have new random values. Furthermore, those 32 weights in the first network might not be in the same locations in the second network if the random seed is maintained between only the two initial weight sets.

To compare architectures, we must perform several training runs and average the final results. Because these extra training runs add to the total runtime of the program, excessive numbers of runs will quickly become impractical. It can also be beneficial to choose a training algorithm that is deterministic (one that does not use random numbers). The experiments that we will perform in this chapter will use five training runs and the resilient propagation (RPROP) training algorithm. RPROP is deterministic, and five runs are an arbitrary choice that provides a reasonable level of consistency. Using the Xavier weight initialization algorithm, introduced in Chapter 4, "Feedforward Neural Networks," will also help provide consistent results.

14.2 Training Parameters

Training algorithms themselves have parameters that you must tune. We don't consider the parameters related to training as hyper-parameters because they are not evident after a neural network has been trained. You can examine a trained neural network to determine easily what hyper-parameters are present. A simple examination of the network reveals the neuron counts and activation function in use. However, determining training parameters such as learning rate and momentum is not possible. Both training parameters and hyper-parameters greatly affect the error rates that the neural network can obtain. However, we can use training parameters only during the actual training.

The three most common training parameters for neural networks are listed here:

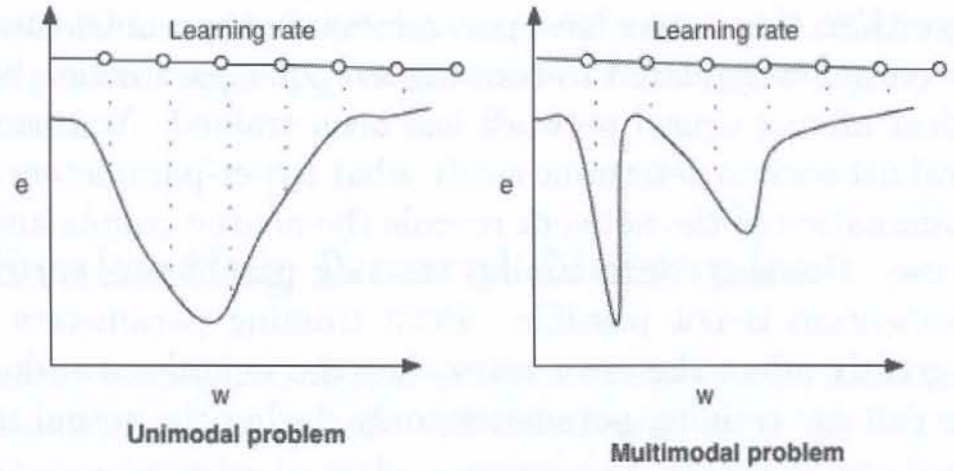
- Learning Rate
- Momentum
- Batch Size

Not all learning algorithms have these parameters. Additionally, you can vary the values chosen for these parameters as learning progresses. We discuss these training parameters in the subsequent sections.

14.2.1 Learning Rate

The learning rate allows us to determine how far each iteration of training will take the weight values. Some problems are very simple to solve, and a high training rate will yield a quick solution. Other problems are more difficult, and a quick learning might disregard a good solution. Other than the runtime of your program, there is no disadvantage in choosing a small learning rate. Figure 14.1 shows how a learning rate might fare on both a simple (unimodal) and complex (multimodal) problem:

Figure 14.1: Learning Rates



The above two charts show the relationship between weight and the score of a network. As the program increases or decreases a single weight, the score changes. A unimodal problem is typically easy to solve because its graph has only one bump, or optimal solution. In this case, we consider a good score to be a low error rate.

A multimodal problem has many bumps, or possible good solutions. If the problem is simple (unimodal), a fast learning rate is optimal because you can charge up the hill to a great score. However, haste makes waste on the second chart, as the learning rate fails to find the two optimums.

Kamiyama, Iijima, Taguchi, Mitsui, et al. (1992) stated that most literature use a learning rate of 0.2 and a momentum of 0.9. Often this learning rate and momentum can be good starting points. In fact, many examples do use these values. The researchers suggest that Equation 14.1 has a strong likelihood of attaining better results.

$$\epsilon = K(1 - \alpha) \tag{14.1}$$

The variable α (alpha) is the momentum; ϵ (epsilon) is the learning rate, and K is a constant related to the hidden neurons. Their research suggests that the tuning of momentum (discussed in the next section) and learning rate are related. We define the constant K by the number of hidden neurons. Smaller numbers of hidden neurons should use a larger K. In our own experimentations, we do not use the equation directly because it is difficult to choose a concrete

value of K. The following calculations show several learning rates based on learning rate and K.

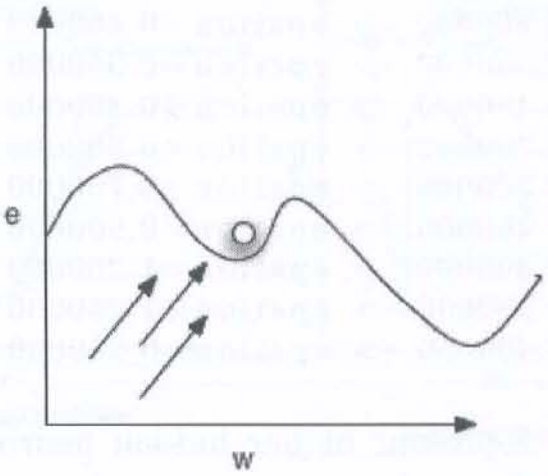
```
k=0.500000, alpha=0.200000 -> epsilon=0.400000
k=0.500000, alpha=0.300000 -> epsilon=0.350000
k=0.500000, alpha=0.400000 -> epsilon=0.300000
k=1.000000, alpha=0.200000 -> epsilon=0.800000
k=1.000000, alpha=0.300000 -> epsilon=0.700000
k=1.000000, alpha=0.400000 -> epsilon=0.600000
k=1.500000, alpha=0.200000 -> epsilon=1.200000
k=1.500000, alpha=0.300000 -> epsilon=1.050000
k=1.500000, alpha=0.400000 -> epsilon=0.900000
```

The lower values of K represent higher hidden neuron counts; therefore the hidden neuron count is decreasing as you move down the list. As you can see, for all momentums (α, alpha) of 0.2, the suggested learning rate (ϵ, epsilon) increases as the hidden neuron counts decrease. The learning rate and momentum have an inverse relationship. As you increase one, you should decrease the other. However, the hidden neuron count controls how quickly momentum and learning rate should diverge.

14.2.2 Momentum

Momentum is a learning property that causes the weight change to continue in its current direction, even if the gradient indicates that the weight change should reverse direction. Figure 14.2 illustrates this relationship:

Figure 14.2: Momentum and a Local Optima



A positive gradient encourages the weight to decrease. The weight has followed the negative gradient up the hill but now has settled into a valley, or a local optima. The gradient now moves to 0 and positive as the other side of the local optima is hit. Momentum allows the weight to continue in this direction and possibly escape from the local-optima valley and possibly find the lower point to the right.

To understand exactly how learning rate and momentum are implemented, recall Equation 6.6, from Chapter 6, "Backpropagation Training," that is repeated as Equation 14.2 for convenience:

$$\Delta w_{(t)} = -\epsilon \frac{\partial E}{\partial w_{(t)}} + \alpha \Delta w_{(t-1)} \tag{14.2}$$

This equation shows how we calculate the change in weight for training iteration t. This change is the sum of two terms that are each governed by the learning rate ϵ (epsilon) and momentum α (alpha). The gradient is the weight's partial derivative of the error rate. The sign of the gradient determines if we should increase or decrease the gradient. The learning rate simply tells backpropagation the percentage of this gradient that the program should apply to the weight change. The program always applies this change to the original weight and then retains it for the next iteration. The momentum α (alpha) subsequently determines the percentage of the previous iteration's weight change that the program should apply to this iteration. Momentum

allows the previous iteration's weight change to carry through to the current iteration. As a result, the weight change maintains its direction. This process essentially gives it "momentum."

Jacobs (1988) discovered that learning rate should be decreased as training progresses. Additionally, as previously discussed, Kamiyama, et al. (1992) asserted that momentum should be increased as the learning rate is decayed. A decreasing learning rate, coupled with an increasing momentum, is a very common pattern in neural network training. The high learning rate allows the neural network to begin exploring a larger area of the search space. Decreasing the learning rate forces the network to stop exploring and begin exploiting a more local region of the search space. Increasing momentum at this point helps guard against local minima in this smaller search region.

14.2.3 Batch Size

The batch size specifies the number of training set elements that you must calculate before the weights are actually updated. The program sums all of the gradients for a single batch before it updates the weights. A batch size of 1 indicates that the weights are updated for each training set element. We refer to this process as online training. The program often sets the batch size to the size of the training set for full batch training.

A good starting point is a batch size equal to 10% of the entire training set. You can increase or decrease the batch size to see its effect on training efficiency. Usually a neural network will have vastly fewer weights than training set elements. As a result, cutting the batch size by a half, or even a fourth, will not have a drastic effect on the runtime of an iteration in standard backpropagation.

14.3 General Hyper-Parameters

In addition to the training parameters just discussed, we must also consider the hyper-parameters. They are significantly more important than training parameters because they determine the neural networks ultimate learning capacity. A neural network with a reduced learning capacity cannot overcome this deficiency with further training.

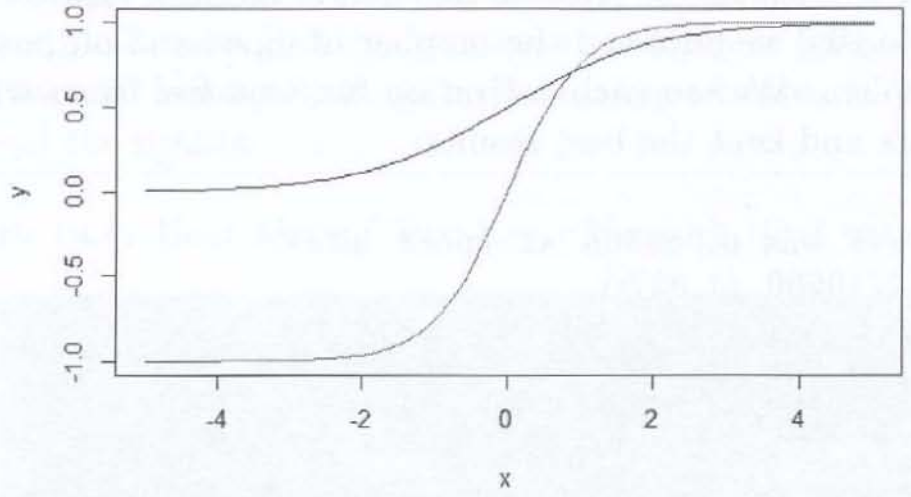
14.3.1 Activation Functions

Currently, the program utilizes two primary types of activation functions inside of a neural network:

- Sigmoidal: Logistic (sigmoid) & Hyperbolic Tangent (tanh)
- Linear: ReLU

The sigmoidal (s-shaped) activation functions have been a mainstay of neural networks, but they are now losing ground to the ReLU activation function. The two most common s-shaped activation functions are the namesake sigmoid activation function and the hyperbolic tangent activation function. The name can cause confusion because sigmoid refers both to an actual activation function and to a class of activation functions. The actual sigmoid activation function has a range between 0 and 1, whereas the hyperbolic tangent function has a range of -1 and 1. We will first tackle hyperbolic tangent versus sigmoid (the activation function). Figure 14.3 shows the overlay of these two activations:

Figure 14.3: Sigmoid and Tanh



As you can see from the figure, the hyperbolic tangent stretches over a much larger range than tanh. Your choice of these two activations will affect the way that you normalize your data. If you are using hyperbolic tangent at the output layer of your neural network, you must normalize the expected outcome between -1 and 1. Similarly, if you are using the sigmoid function for the output layer, you must normalize the expected outcome between -1 and 1. You should normalize the input to -1 to 1 for both of these activation functions. The x-values (input) above +1 will saturate to +1 (y-values) for both sigmoid and hyperbolic tangent. As x-values go below -1, the sigmoid activation function saturates to y-values of 0, and hyperbolic tangent saturates to y-values of -1.

The saturation of sigmoid to values of 0 in the negative direction can be problematic for training. As a result, Kalman & Kwasny (1992) recommend hyperbolic tangent in all situations instead of sigmoid. This recommendation corresponds with many literature sources. However, this argument only extends to the choice between sigmoidal activation functions. A growing body of recent research favors the ReLU activation function in all cases over the sigmoidal activation functions.

Zeiler et al. (2014), Maas, Hannun, Awni & Ng (2013) and Glorot, Bordes & Bengio (2013) all recommend the ReLU activation function over its sigmoidal counterparts. "Chapter 9, "Deep Learning," includes the advantages

of the ReLU activation function. In this section, we will examine an experiment that compares the ReLU to the sigmoid, we used a neural network with a hidden layer of 1,000 neurons. We ran this neural network against the MNIST data set. Obviously, we adjusted the number of input and output neurons to match the problem. We ran each activation function five times with different random weights and kept the best results:

```
Sigmoid:
Best valid loss was 0.068866 at epoch 43.
Incorrect 192/10000 (1.92%)
ReLU:
Best valid loss was 0.068229 at epoch 17.
Incorrect 170/10000 (1.7000000000000002%)
```

The accuracy rates for each of the above neural networks on a validation set. As you can see, the ReLU activation function did indeed have the lowest error rate and achieved it in fewer training iterations/epochs. Of course, these results will vary, depending on the platform used.

14.3.2 Hidden Neuron Configurations

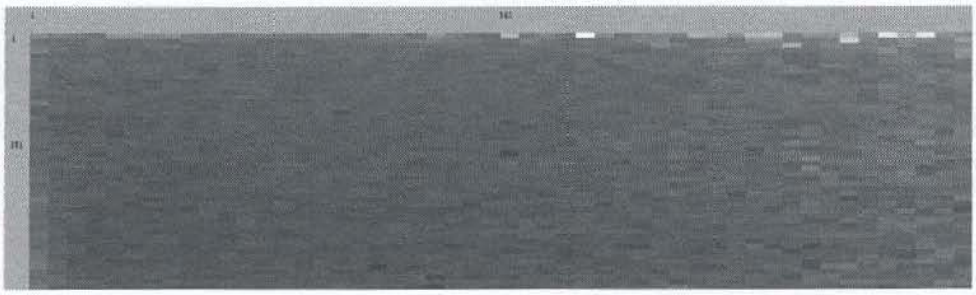
Hidden neuron configurations have been a frequent source of questions. Neural network programmers often wonder exactly how to structure their networks. As of the writing of this book, a quick scan of Stack Overflow shows over 50 questions related to hidden neuron configurations. You can find the questions at the following link:

http://goo.gl/ruWpcb

Although the answers may vary, most of them simply advise that the programmer "experiment and find out." According to the universal approximation theorem, a single-hidden-layer neural network can theoretically learn any pattern (Hornik, 1991). Consequently, many researchers suggest only single-hidden-layer neural networks. Although a single-hidden-layer neural network can learn any pattern, the universal approximation theorem does not state that this process is easy for a neural network. Now that we have efficient techniques to train deep neural networks, the universal approximation theorem is much less important.

To see the effects of hidden neurons and neuron counts, we will perform an experiment that will look at one-layer and two-layer neural networks. We will try every combination of hidden neurons up to two 50-neuron layers. This neural network will use a ReLU activation function and RPROP. This experiment took over 30 hours to run on an Intel I7 quad-core. Figure 14.4 shows a heat map of the results:

Figure 14.4: Heat Map of Two-Layer Network (first experiment)



The best configuration reported by the experiment was 35 neurons in hidden layer 1, and 15 neurons in hidden layer 2. The results of this experiment will vary when repeated. The above diagram shows the best-trained networks in the lower-left corner, as indicated by the darker squares. This indicates that the network favors a large first hidden layer with smaller second hidden layers. The heat map shows the relationships between the different configurations. We achieved better results with smaller neuron counts on the second hidden layer. This occurred because the neuron counts constricted the information flow to the output layer. This approach is consistent with research into auto-encoders in which successively smaller layers force the neural network to generalize information, rather than overfit. In general, based on the experiment here, we advise using at least two hidden layers with successively smaller layers.

14.4 LeNet-5 Hyper-Parameters

The LeNet-5 convolutional neural networks introduce additional layer types that bring more choices in the construction of neural networks. Both the convolutional and max-pooling layers create other choices for hyper-parameters. Chapter 10, "Convolutional Neural Networks" contains a complete list of

hyper-parameters that the LeNet-5 network introduces. In this section, we will review LeNet-5 architectural recommendations recently suggested in scientific papers.

Most literature on LeNet-5 networks supports the use of a max-pool layer to follow every convolutional layer. Ideally, several convolutional/max-pool layers reduce the resolution at each step. Chapter 6, "Convolutional Neural Networks" includes this demonstration. However, very recent literature seems to indicate that max-pool layers should not be used at all.

On November 7, 2014, the website Reddit featured Dr. Geoffrey Hinton for an "ask me anything (AMA)" session. Dr. Hinton is the foremost researcher in deep learning and neural networks. During the AMA session, Dr. Hinton was asked about max-pool layers. You can read his complete response here:

https://goo.gl/TgBakL

Overall, Dr. Hinton begins his answer saying, "The pooling operation used in convolutional neural networks is a big mistake, and the fact that it works so well is a disaster." He then proceeds with a technical description of why you should never use max-pooling. At the time of this book's publication, his response is fairly recent and controversial. Therefore we suggest that you try the convolutional neural networks both with and without max-pool layers, as their future looks uncertain.

14.5 Chapter Summary

Selecting a good set of hyper-parameters is one of the most difficult tasks for the neural network programmer. The number of hidden neurons, activation functions, and layer structures are all examples of neural network hyper-parameters that the programmer must adjust and fine-tune. All these hyper-parameters can affect the overall capacity of the neural network to learn patterns. As a result, you must choose them correctly.

Most current literature suggests using the ReLU activation function in place of the sigmoidal (s-shaped) activation functions. If you are going to use a sigmoidal activation, most literature recommends that you use the hyperbolic

tangent activation function instead of the sigmoidal activation function. The ReLU activation function is more compatible with deep neural networks.

The number of hidden layers and neurons is also an important hyper-parameter for neural networks. It is generally advisable that successive hidden layers contain a smaller number of neurons than the immediately previous layer. This adjustment has the effect of constraining the data from the inputs and forcing the neural network to generalize and not memorize, which results in overfitting.

We do not consider training parameters as hyper-parameters because they do not affect the structure of the neural network. However, you still must choose a proper set of training parameters. The learning rate and momentum are two of the most common training parameters for neural networks. Generally, you should initially set the learning rate high and decrease it as training continues. You should move the momentum value inversely with the learning rate.

In this chapter, we examined how to structure neural networks. While we provided some general recommendations, the data set generally drives the architecture of the neural network. Consequently, you must analyze the data set. We will introduce the t-SNE dimension reduction algorithm in the next chapter. This algorithm will allow you to visualize graphically your data set and see issues that occur while you are creating a neural network for that data set.