Chapter 8: Pruning Neural Networks

- What is Pruning?
- Incremental Pruning
- Selective Pruning
- Pruning Examples

In chapters 6 and 7, we saw that you can use simulated annealing and genetic algorithms to train neural networks. These two techniques employ various algorithms to better fit the weights of a neural network to the problem to which it is being applied. However, these techniques do nothing to adjust the structure of the neural network.

In this chapter, we will examine two algorithms that can be used to actually modify the structure of a neural network. This structural modification will not generally improve the error rate of the neural network, but it can make the neural network more efficient. The modification is accomplished by analyzing how much each neuron contributes to the output of the neural network. If a particular neuron's connection to another neuron does not significantly affect the output of the neural network, the connection will be pruned. Through this process, connections and neurons that have only a marginal impact on the output are removed.

This process is called pruning. In this chapter, we will examine how pruning is accomplished. We will begin by examining the pruning process in greater detail and will discuss some of the popular methods. Finally, this chapter will conclude by providing two examples that demonstrate pruning.

Understanding Pruning

Pruning is a process used to make neural networks more efficient. Unlike genetic algorithms or simulated annealing, pruning does not increase the effectiveness of a neural network. The primary goal of pruning is to decrease the amount of processing required to use the neural network.

Pruning can be especially effective when performed on a large neural network that is taking too long to execute. Pruning works by analyzing the connections of the neural network. The pruning algorithm looks for individual connections and neurons that can be removed from the neural network to make it operate more efficiently. By pruning unneeded connections, the neural network can be made to execute faster. This allows the neural network to perform more work in a given amount of time. In the next two sections we will examine how to prune both connections and neurons.

Pruning Connections

Connection pruning is central to most pruning algorithms. The individual connections between the neurons are analyzed to determine which connections have the least impact on the effectiveness of the neural network. One of the methods that we will examine will remove all connections that have a weight below a certain threshold value. The second method evaluates the effectiveness of the neural network as certain weights are considered for removal. Connections are not the only thing that can be pruned. By analyzing which connections were pruned, we can also prune individual neurons.

Pruning Neurons

Pruning focuses primarily on the connections between the individual neurons of the neural network. However, individual neurons can be pruned as well. One of the pruning algorithms that we will examine later in this chapter is designed to prune neurons as well as connections.

To prune individual neurons, the connections between each neuron and the other neurons must be examined. If one particular neuron is surrounded entirely by weak connections, there is no reason to keep that neuron. If we apply the criteria discussed in the previous section, we can end up with neurons that have no connections. This is because all of the neuron's connections were pruned. Such a neuron can then be pruned itself.

Improving or Degrading Performance

It is possible that pruning a neural network may improve its performance. Any modifications to the weight matrix of a neural network will always have some impact on the accuracy of the recognitions made by the neural network. A connection that has little or no impact on the neural network may actually be degrading the accuracy with which the neural network recognizes patterns. Removing such a weak connection may improve the overall output of the neural network.

Unfortunately, it is also possible to decrease the effectiveness of the neural network through pruning. Thus, it is always important to analyze the effectiveness of the neural network before and after pruning. Since efficiency is the primary benefit of pruning, you must be careful to evaluate whether an improvement in the processing time is worth a decrease in the neural network's effectiveness. The program in the example that we will examine later in this chapter will evaluate the overall effectiveness of the neural network both before and after pruning. This will give us an idea of what effect the pruning process had on the effectiveness of the neural network.

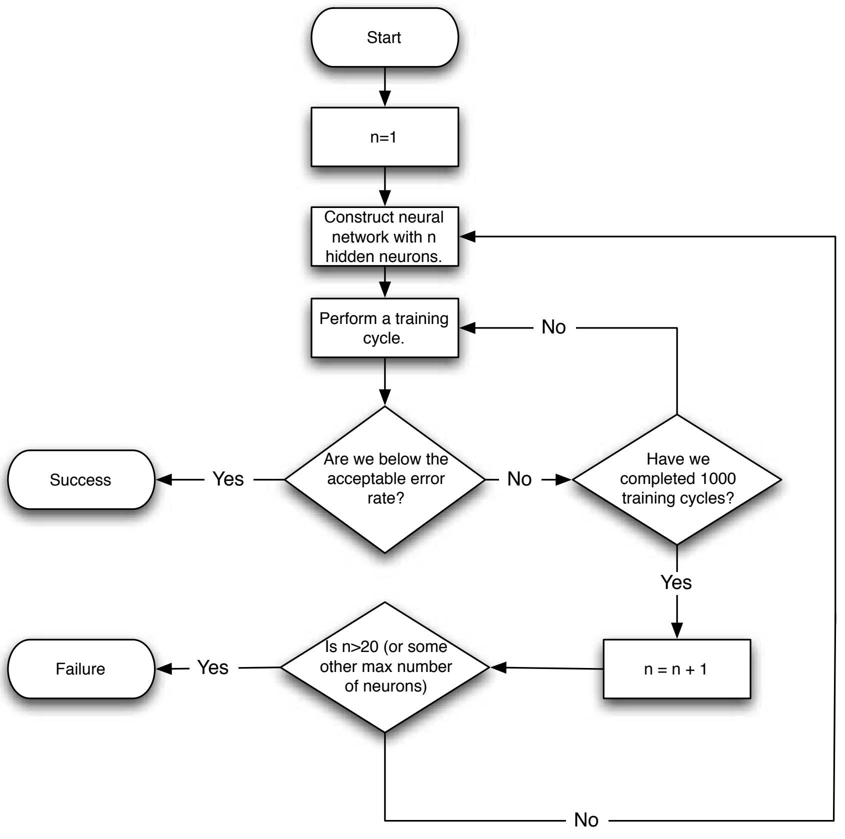
Pruning Algorithms

We will now review exactly how pruning takes place. In this section we will examine two different methods for pruning. These two methods work in somewhat opposite ways. The first method, incremental pruning, works by gradually increasing the number of hidden neurons until an acceptable error rate has been obtained. The second method, selective pruning, works by taking an existing neural network and decreasing the number of hidden neurons as long as the error rate remains acceptable.

Incremental Pruning

Incremental pruning is a trial and error approach to finding an appropriate number of hidden neurons. This method is summarized in Figure 8.1.

Figure 8.1: Flowchart of the incremental pruning algorithm.


Start
n=1
Construct neural network with n hidden neurons.
Perform a training cycle.
No
Success
Yes
Are we below the acceptable error rate?
No
Have we completed 1000 training cycles?
Yes
Failure
Yes
Is n>20 (or some other max number of neurons)
n = n + 1
No

The incremental pruning algorithm begins with an untrained neural network. It then attempts to train the neural network many times. Each time, it uses a different set of hidden neurons.

The incremental training algorithm must be supplied with an acceptable error rate. It is looking for the neural network with the fewest number of hidden neurons that will cause the error rate to fall below the desired level. Once a neural network that can be trained to fall below this rate is found, the algorithm is complete.

As you saw in chapter 5, "Feedforward Backpropagation Neural Networks," it is often necessary to train for many cycles before a solution is found. The incremental pruning algorithm requires the entire training session to be completed many times. Each time a new neuron is added to the hidden layer, the neural network must be retrained. As a result, it can take a long time for the incremental pruning algorithm to run.

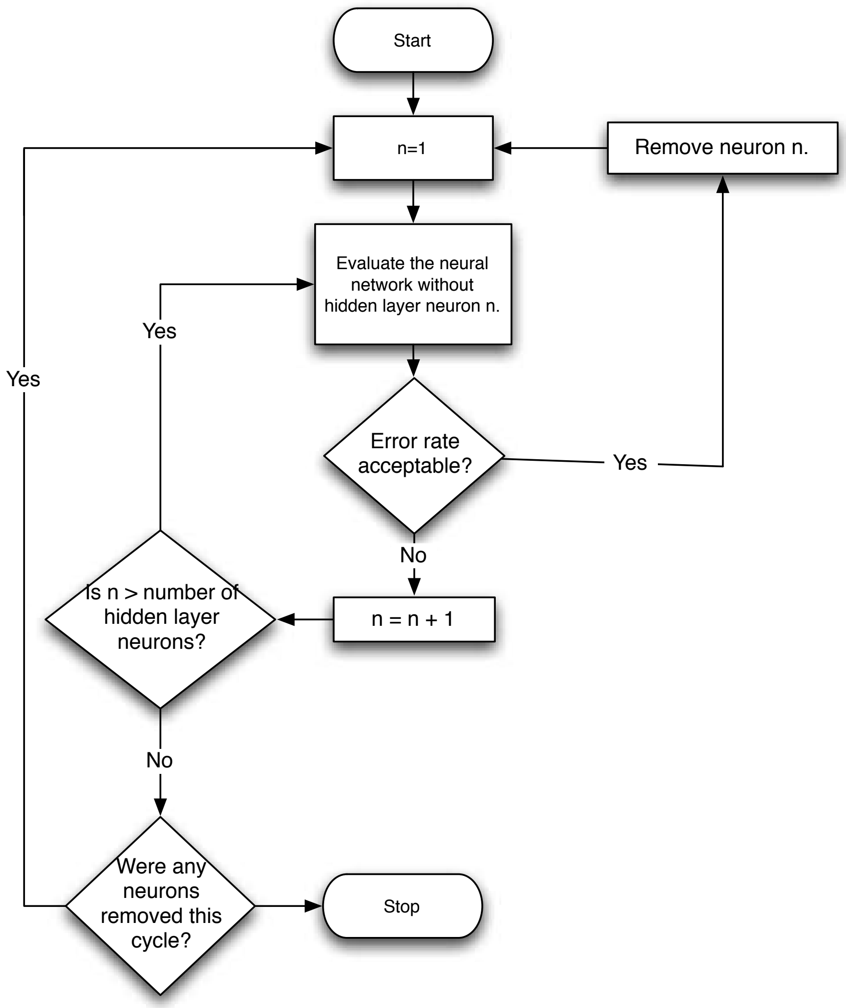
The neural network will train for different numbers of hidden neurons, beginning initially with a single neuron. Because the error rate does not drop sufficiently fast, the single hidden neuron neural network will quickly be abandoned. Any number of methods can be used to determine when to abandon a neural network. The method that will be used in this chapter is to check the current error rate after intervals of 1,000 cycles. If the error does not decrease by a single percentage point, then the search will be abandoned. This allows us to quickly abandon hidden layer sizes that are too small for the intended task.

One advantage of the incremental pruning algorithm is that it will usually create neural networks with fewer hidden neurons than the other methods. The biggest disadvantage is the amount of processor time that it takes to run this algorithm. Now that you have been introduced to the incremental pruning algorithm, we will examine the selective pruning algorithm.

Selective Pruning

The selective pruning algorithm differs from the incremental pruning algorithm in several important ways. One of the most notable differences is the beginning state of the neural network. No training was required before beginning the incremental pruning algorithm. This is not the case with the selective pruning algorithm. The selective pruning algorithm works by examining the weight matrixes of a previously trained neural network. The selective training algorithm will then attempt to remove neurons without disrupting the output of the neural network. The algorithm used for selective pruning is shown in Figure 8.2.

Figure 8.2: Flowchart of the selective pruning algorithm.


Start
n=1
Remove neuron n.
Evaluate the neural network without hidden layer neuron n.
Yes
Yes
Error rate acceptable?
Yes
No
Is n > number of hidden layer neurons?
n = n + 1
No
Were any neurons removed this cycle?
Stop

As you can see, the selective pruning algorithm is something of a trial and error approach. The selective pruning algorithm attempts to remove neurons from the neural network until no more neurons can be removed without degrading the performance of the neural network.

To begin this process, the selective pruning algorithm loops through each of the hidden neurons. For each hidden neuron encountered, the error level of the neural network is evaluated both with and without the specified neuron. If the error rate jumps beyond a predefined level, the neuron will be retained and the next neuron will be evaluated. If the error rate does not jump by much, the neuron will be removed.

Once the program has evaluated all neurons, the program repeats the process. This cycle continues until the program has made one pass through the hidden neurons without removing a single neuron. Once this process is complete, a new neural network is achieved that performs acceptably close to the original, yet has fewer hidden neurons.

The major advantage of the selective pruning algorithm is that it takes very little processing time to complete. The program in the example presented later in this chapter requires under one second to prune a neural network with 10 hidden layer neurons. This is considerably less time than the incremental pruning algorithm described in the previous section.

Implementing Pruning

Now that it has been explained how the pruning algorithms work, you will be shown how to implement them in Java. In this section, we will examine a general class that is designed to prune feedforward backpropagation neural networks. The name of this class is simply "Prune." This class accepts a "Network" class and performs either incremental or selective pruning. We will begin by examining the prune class. We will then examine the incremental and selective algorithms within the class.

The Prune Class

The **`Prune`** class contains all of the methods and properties that are required to prune a feedforward backpropagation neural network. There are several properties that are used internally by many of the methods that make up the **`Prune`** class. These properties are summarized in Table 8.1.

Table 8.1: Variables Used for the Prune Process

Variable	Purpose
backprop	The backpropagation object to be pruned.
cycles	The number of cycles.
done	Flag to indicate if the incremental pruning process is done or not.
error	The current error.
ideal	The ideal results from the training set.
hiddenNeuronCount	The number of hidden neurons.
markErrorRate	Used to determine if training is still effective. Holds the error level determined in the previous 1000 cycles. If no significant drop in error occurs for 1000 cycles, training ends.
maxError	The maximum acceptable error.
momentum	The desired momentum.
rate	The desired learning rate (for backpropagation).
sinceMark	Used with markErrorRate. This is the number of cycles since the error was last marked.
train	The training set.

You will now be shown how the selective and incremental pruning algorithms are implemented. We will begin with incremental pruning.

Incremental Pruning

As you will recall from earlier in this chapter, the process of incremental pruning involves increasing the number of neurons in the hidden layer until the neural network is able to be trained sufficiently well. This should automatically lead us to a good number of hidden neurons. The constructor used to implement incremental pruning is very simple. It collects the required parameters and stores them in the class's properties.

The parameters required by the incremental pruning constructor are the usual parameters needed to perform backpropagation training of a feedforward neural network. The learning rate and momentum are both required. These backpropagation constants are discussed in greater detail in chapter 5. Additionally, a training set, along with the ideal outputs for the training set, are also required.

The final parameter that is required by the incremental pruning constructor is the minimum acceptable error. This is the error rate that is sufficient for a neural network to be trained. As the incremental pruning algorithm progresses, it will try to train neural networks with various numbers of hidden neurons. The final number of hidden neurons will be determined by the neural network with the fewest number of hidden neurons that is able to be trained to reach this error level.

The **Prune** class contains two constructors; one is for incremental pruning and the other is for selective pruning. The constructor for the **Prune** class to be used with incremental pruning is shown here:

```
public Prune(final double rate, final double momentum,
  final double train[][], final double ideal[][],
  final double maxError)
```

This provides the class with the training and ideal sets to be used with backpropagation, as well as a maximum acceptable error.

Starting the Incremental Pruning Process

To begin the process of incremental pruning, you must call the **startIncremental** method. This sets up the **Prune** class for a new pruning run. The signature for the **startIncremental** method is shown here.

```
public void startIncremental()
```

First, some of the properties are set to their initial values. One such property is the number of neurons that will exist in the hidden layer. Initially, we begin with one single neuron in the hidden layer.

```
this.hiddenNeuronCount = 1;
this.cycles = 0;
this.done = false;
```

A new **FeedforwardNetwork** object is created to hold the neural network that will be prepared as the training progresses.

```
this.currentNetwork = new FeedforwardNetwork();
```

Next, the layers are added to the network.

```
this.currentNetwork
  .addLayer(new FeedforwardLayer(this.train[0].length));
this.currentNetwork.addLayer(new FeedforwardLayer(
  this.hiddenNeuronCount));
this.currentNetwork.addLayer(new
  FeedforwardLayer(this.ideal[0].length));
this.currentNetwork.reset();
```

A backpropagation object is created to train the network.

```
this.backprop = new Backpropagation(this.currentNetwork, this.
train,
this.ideal, this.rate, this.momentum);
```

Now, the incremental prune algorithm is ready to begin. In the next section, you will see how the main loop of the incremental algorithm is constructed.

Main Loop of the Incremental Algorithm

The incremental pruning algorithm is processor intensive and may take some time to run. This is because the incremental algorithm is literally trying different numbers of hidden neurons, in a trial and error fashion, until a network is identified that has the fewest number of neurons while producing an acceptable error level.

The incremental pruning algorithm is designed so that a background thread can rapidly call the **pruneIncremental** method until the **getDone** method indicates that the algorithm is done. Of course, the **pruneIncremental** method can be called from the main thread, as well. The signature for the **pruneIncremental** method is shown here:

```
public void pruneIncremental()
```

If the work has already been done, then exit.

```
if (this.done) {
  return;
}
```

The **pruneIncremental** method begins by first checking to see if it is already done. If the algorithm is already done, the **pruneIncremental** method simply returns.

The next step is to attempt a single training cycle for the neural network. The program calls **increment**, which loops through all of the training sets and calculates the error based on the ideal outputs.

```
this.backprop.iteration();
```

Once the training set has been presented, the root mean square (RMS) error is calculated. The RMS error was discussed in chapter 4, “How a Machine Learns.” Calculating the RMS error allows the pruning algorithm to determine if the error has reached the desired level.

```
this.error = this.backprop.getError();
this.cycles++;
increment();
```

Each time a training cycle is executed, the neural network must check to see if the number of hidden neurons should be increased or if training should continue with the current neural network.

Incrementing the Number of Neurons

To determine if the number of hidden neurons should be increased or not, the helper method named **increment** is used. The **increment** method keeps track of training progress for the neural network. If the training improvement for each cycle falls below a constant value, further training is deemed futile. In this case, we increment the number of hidden neurons and continue training.

The **increment** method begins by setting a flag that indicates whether or not it should increment the number of hidden neurons. The default is **false**, which means do not increment the number of neurons. The signature for the increment method is shown here:

```
protected void increment()
```

Start by setting the **doit** variable to **false**. This variable is set to **true** if a better neural network configuration is found.

```
boolean doit = false;
```

The algorithm that this class uses to determine if further training is futile, works by examining the amount of improvement every 10,000 cycles. If the error rate does not change by more than one percent within 10,000 cycles, further training is deemed futile and the number of neurons in the hidden layer is incremented.

The following lines of code accomplish this evaluation. First, when the **markErrorRate** is zero, it means that we are just beginning and have not yet sampled the error rate. In this case, we initialize the **markErrorRate** and **sinceMark** variables.

```
if (this.markErrorRate == 0) {
  this.markErrorRate = this.error;
  this.sinceMark = 0;
} else {
```

If the **markErrorRate** is not zero, then we are tracking errors. We should increase the **sinceMark** cycle counter and determine if more than 10,000 cycles have been completed since we last sampled the error rate.

```
this.sinceMark++;
if (this.sinceMark > 10000) {
```

If more than 10,000 cycles have passed, we check to see if the improvement between the **markErrorRate** and current error rate is less than one percent. If this is the case, then we set the flag to indicate that the number of hidden neurons should be incremented.

```
if ((this.markErrorRate - this.error) < 0.01) {
  doit = true;
}
```

The error rate is then sampled again.

```
    this.markErrorRate = this.error;
    this.sinceMark = 0;
  }
}
```

If the error rate is below the acceptable error, then we have found the number of neurons we will recommend for the neural network. We can now set the **`done`** flag to **`true`**.

```
if (this.error < this.minError) {
  this.done = true;
}
```

If the flag is set, we increment the number of neurons as follows:

```
if (doit) {
  this.cycles = 0;
  this.hiddenNeuronCount++;

  this.currentNetwork = new FeedforwardNetwork();
  this.currentNetwork.addLayer(new FeedforwardLayer(
    this.train[0].length));
  this.currentNetwork.addLayer(new FeedforwardLayer(
    this.hiddenNeuronCount));
  this.currentNetwork.addLayer(new FeedforwardLayer(
    this.ideal[0].length));
  this.currentNetwork.reset();

  this.backprop = new Backpropagation(this.currentNetwork,
      this.train, this.ideal, this.rate, this.momentum);
}
```

As you can see, a new neural network is constructed after the number of hidden neurons is increased. Also, the cycle count is reset to zero, because we will begin training a new neural network.

You should now be familiar with how the incremental pruning algorithm is implemented. Later in this chapter, we will construct a sample program that makes use of this algorithm. For now, we will cover the implementation of the second pruning algorithm, the selective pruning algorithm.

Selective Pruning

Now that you have seen how the incremental pruning algorithm was implemented, we will examine the implementation of the selective pruning algorithm. In some ways, the selective pruning algorithm works in reverse of the incremental pruning algorithm. Where the incremental pruning algorithm starts small and grows, the selective pruning algorithm starts with a large, pretrained neural network, and selects neurons for removal.

To use selective pruning, you can make use of a simplified version of the **Prune** constructor. Selective pruning does not need to know anything about the learning rate or momentum of the backpropagation process, because it does not involve the backpropagation algorithm. The simplified version of the constructor is shown below.

```
public Prune(Network network,double train[][],
  double ideal[][], final double maxError)
```

As you can see, you are only required to pass a neural network training set and the ideal results to the selective pruning algorithm. The constructor is very simple and merely stores the values that it was passed. We will now examine the implementation of the selective pruning methods. We will begin by examining the main loop of the algorithm.

Main Loop of the Selective Pruning Algorithm

The main loop of the selective pruning algorithm is much less processor intensive than the incremental pruning algorithm. The incremental pruning algorithm that we examined in the previous section was designed to run as a background thread due to the large number of cycles that might be required to find a solution. This is not the case with the selective pruning algorithm.

The selective pruning algorithm is designed to evaluate the performance of the neural network when each hidden neuron is removed. If the performance of the neural network does not degrade substantially with the removal of a neuron, that neuron will be removed permanently. This process will continue until no additional neurons can be removed without substantially degrading the performance of the neural network. Thus, the selective pruning algorithm is designed to perform the entire algorithm with one method call. There is no reason for a background thread, as this method should be able to perform its task almost instantaneously.

The method that should be called to prune a neural network selectively is the **pruneSelective** method. We will now examine how this method performs. The signature for this method is shown here:

```
public int pruneSelective()
```

Following is the body of the **pruneSelective** method.

```
final int i = this.getHiddenCount();
while (findNeuron()) {
  ;
}
return (i - this.getHiddenCount());
```

As you can see from the above code, the current number of hidden neurons is stored in the variable **i** for future use. We then enter a loop and iterate until no additional neurons can be removed. Finally, the **pruneSelective** method returns the number of neurons that were removed from the neural network. The new optimized neural network is stored in the **currentNetwork** property of the **Prune** class and can be accessed using the **getCurrentNetwork** method.

The real work performed by the selective pruning algorithm is done by the **findNeuron** method. It is the **findNeuron** method that actually identifies and removes neurons that do not have an adverse effect on the error rate. The **findNeuron** method will remove a neuron, if possible, and return **true**. If no neuron can be removed, the **findNeuron** method returns a value of **false**. As you can see from the above code in the **pruneSelective** method, the **findNeuron** method is called as long as the return value is **true**. The signature for the **findNeuron** method is shown here:

```
protected boolean findNeuron()
```

We will now examine the contents of the **findNeuron** method. This method begins by calculating the current error rate.

```
for (int i = 0; i < this.currentNetwork.getHiddenLayerCount();
       i++) {
```

The error rate is then recalculated, and we evaluate the effect this has on the quality of the neural network. As long as the quality is still below **maxError**, we continue pruning.

```
  final FeedforwardNetwork trial = this.clipHiddenNeuron(i);
  final double e = determineError(trial);
  if (e < this.maxError) {
```

If we have an acceptable network when the neuron is removed, we exit the method and **true** is returned. The value of **true** indicates that the neuron was removed, and we should attempt to remove additional neurons.

```
    this.currentNetwork = trial;
    return true;
  }
}
```

If no neurons were removed, **false** is returned. This indicates that further processing with be of no additional value.

```
return false;
```

You will now be shown the process by which neurons are removed from the neural network.

Removing Neurons

Removing a neuron from a **FeedforwardLayer** object is a relatively straightforward process. The task of removing an individual hidden neuron is encapsulated inside the **Prune** method of the **FeedforwardLayer** class in the **clipHiddenNeuron** method. The first thing that the **clipHiddenNeuron** method does is create a new neural network with one less neuron.

```
public void prune(final int neuron) {
```

First, it deletes the row of this layer's matrix that corresponds to the targeted neuron.

```
if (this.matrix != null) {
  setMatrix(MatrixMath.deleteRow(this.matrix, neuron));
}
```

Next, it deletes the column on the previous layer's matrix that corresponds to the targeted neuron.

```
final FeedforwardLayer previous = this.getPrevious();
if (previous != null) {
  if (previous.getMatrix() != null) {
      previous.setMatrix(
    MatrixMath.deleteCol(previous.getMatrix(), neuron));
}
```

This completes the process for removing the targeted neuron and concludes how the **Prune** class is constructed. We covered both incremental and selective pruning. We will now see how to make use of this class in a Java program.

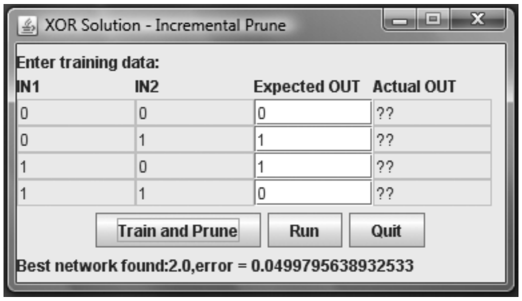
Using the Prune Class

In this section, we will examine two programs that are provided as examples to demonstrate the use of the **Prune** class. These examples demonstrate the two pruning methods that we have discussed in this chapter. We will begin with the incremental pruning example.

The Incremental Pruning Example

This example is based on the XOR problem that was presented in chapter 5. Thus, the complete code for this program will not be shown here. Rather, the additional code that was added to the XOR example in chapter 5 to support pruning is shown. You can see the output from the incremental pruning example in Figure 8.3.

Figure 8.3: The incremental pruning example.



Enter training data:

IN1	IN2	Expected OUT	Actual OUT
0	0	0	??
0	1	1	??
1	0	1	??
1	1	0	??

Best network found:2.0,error = 0.0499795638932533

When the "Prune/Train" button is pressed, a background thread is created. This background thread executes the prune example's **run** method. We will now examine the contents of the **run** method.

The signature for the **run** method is shown here:

```
public void run()
```

The **run** method begins by reading any data entered by the user. Both the training sets and the ideal results are read from the window.

```
final double xorData[][] = getGrid();
final double xorIdeal[][] = getIdeal();
int update = 0;
```

Next, a **Prune** object is created named **prune**. This object will use a learning rate of 0.7 and a momentum of 0.5. The **startIncremental** method is called to prepare for incremental pruning.

```
final Prune prune = new Prune(0.7, 0.5, xorData, xorIdeal, 0.05);
prune.startIncremental();
```

As long as the pruning algorithm is not done, the loop will continue.

```
while (!prune.getDone()) {
```

One iteration of incremental training will be performed.

```
  prune.pruneIncremental();
  update++;
```

The screen is updated every ten iterations.

```
  if (update == 10) {
    this.status.setText("Cycles:" + prune.getCycles()
      + ",Hidden Neurons:" + prune.getHiddenNeuronCount()
      + ", Current Error=" + prune.getError());
      update = 0;
  }
}
```

Finally, the system reports the best network found.

```
this.status.setText("Best network found:"
+ prune.getHiddenNeuronCount()
+ ",error = " + prune.getError());
```

This network is then copied to the current network.

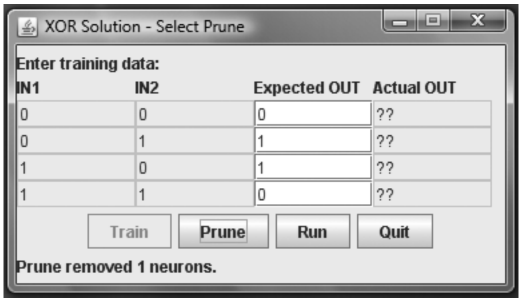
```
this.network = prune.getCurrentNetwork();
this.btnRun.setEnabled(true);
```

In this section, you have seen how the **`Prune`** class was used to implement an incremental pruning of a neural network. In the next section, you will see how to use the **`Prune`** class for a selective pruning of a neural network.

The Selective Pruning Example

This selective pruning example is also based on the XOR problem that was shown in chapter 5; thus, the complete code is not shown here. Rather, only the additional code that was added to the XOR example in chapter 5 to support selective pruning is presented. You can see the output from the incremental pruning example in Figure 8.4.

Figure 8.4: The selective pruning example.



Enter training data:

IN1	IN2	Expected OUT	Actual OUT
0	0	0	??
0	1	1	??
1	0	1	??
1	1	0	??

Prune removed 1 neurons.

Because the selective pruning example requires that a neural network already be present to prune, you should begin by clicking the “Train” button. This will train the neural network using the backpropagation method. The neural network shown in this example initially contains ten neurons in the hidden layer. Clicking the “Prune” button will begin a selective pruning and attempt to remove some of the neurons from the hidden layer.

The code necessary to implement a selective pruning algorithm is very simple. Because this algorithm executes very quickly, there is no need for a background thread. This greatly simplifies the use of the selective pruning algorithm. When the "Prune" button is clicked, the **prune** method is executed. The signature for the **prune** method is shown here:

```
public void prune()
```

The **prune** method begins by obtaining the data from the grid.

```
final double xorData[][] = getGrid();
final double xorIdeal[][] = getIdeal();

final Prune prune = new Prune(this.network, xorData, xorIde-
al,0.05);
final int count = prune.pruneSelective();
this.network = prune.getCurrentNetwork();
this.status.setText("Prune removed " + count + " neurons.");
this.btnTrain.setEnabled(false);
```

Next, a **Prune** object is instantiated. The **pruneSelective** method is called which returns a count of the neurons that were removed during the prune. This value is displayed to the user and the new network is copied to the current network. The user may now run the network to see, first hand, the performance of the neural network. You will find that the selective pruning algorithm is usually able to eliminate two or three neurons. While this leaves more neurons than the incremental algorithm, significantly less processor time is required, and there is no need to retrain the neural network.

Chapter Summary

As you learned in this chapter, it is possible to prune neural networks. Pruning a neural network removes connections and neurons in order to make the neural network more efficient. The goal of pruning is not to make the neural network more effective at recognizing patterns, but to make it more efficient. There are several different algorithms for pruning a neural network. In this chapter, we examined two of these algorithms.

The first algorithm we examined was called the incremental pruning algorithm. This algorithm trains new neural networks as the number of hidden neurons is increased. The incremental pruning algorithm eventually settles on the neural network that has the fewest neurons in the hidden layer, yet still maintains an acceptable error level. While the incremental algorithm will often find the ideal number of hidden neurons, in general, it takes a considerable amount of time to execute.

The second algorithm we examined was called the selective pruning algorithm. The selective pruning algorithm begins with a neural network that has already been trained. The algorithm then removes hidden neurons as long as the error stays below a specified level. Although the selective pruning algorithm will often not find the optimal number of neurons it will execute considerably faster than the incremental algorithm.

The primary goal of neural network pruning is to improve efficiency. By pruning a neural network, you are able to create a neural network that will execute faster and require fewer processor cycles. If your neural network is already operating sufficiently fast, you must evaluate whether the pruning is justified. Even when efficiency is of great importance, you must weigh the trade-offs between efficiency and a reduction in the effectiveness of your neural network.

Thus far, this book has focused primarily on using neural networks to recognize patterns. Neural networks can also be taught to predict future trends. By providing a neural network with a series of time-based values, it can predict subsequent values. The next two chapters will focus on predictive neural networks. This type of neural network is sometimes called a temporal neural network.

Vocabulary

Connection Significance

Incremental Pruning

Pruning

Selective Pruning

Questions for Review

1. Describe the steps in the incremental pruning method.

2. Describe the steps in the selective pruning method.

3. You have a neural network that has already been trained and produces good results. However, you wonder if you might be able to remove several hidden neurons. Which pruning method would you use to accomplish this? Why?

4. You would like to remove a neuron from the hidden layer. What, if anything, must be done to the weight matrix between the input layer and the hidden layer? What, if anything, must be done to the weight matrix between the hidden layer and the output layer?

5. Which pruning method must use one of the learning algorithms as part of its process.