Chapter 11

Pruning and Model Selection

- Pruning a Neural Network
- Model Selection
- Random vs. Grid Search

In previous chapters, we learned that you could better fit the weights of a neural network with various training algorithms. In effect, these algorithms adjust the weights of the neural network in order to lower the error of the neural network. We often refer to the weights of a neural network as the parameters of the neural network model. Some machine learning models might have parameters other than weights. For example, logistic regression (which we discussed in *Artificial Intelligence for Humans, Volume 1*) has coefficients as parameters.

When we train the model, the parameters of any machine learning model change. However, these models also have hyper-parameters that do not change during training algorithms. For neural networks, the hyper-parameters specify the architecture of the neural network. Examples of hyper-parameters for neural networks include the number of hidden layers and hidden neurons.

In this chapter, we will examine two algorithms that can actually modify or suggest a structure for the neural network. Pruning works 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.

The other algorithm that we introduce in this chapter is model selection. While pruning starts with an already trained neural network, model selection creates and trains many neural networks with different hyper-parameters. The program then selects the hyper-parameters producing the neural network that achieves the best validation score.

11.1 Understanding Pruning

Pruning is a process that makes neural networks more efficient. Unlike the training algorithms already discussed in this book, pruning does not increase the training error of the neural network. The primary goal of pruning is to decrease the amount of processing required to use the neural network. Additionally, pruning can sometimes have a regularizing effect by removing complexity from the neural network. This regularization can sometimes decrease the amount of overfitting in the neural network. This decrease can help the neural network perform better on data that were not part of the training set.

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 and possibly decrease overfitting. In the next two sections, we will examine how to prune both connections and neurons.

11.1.1 Pruning Connections

Connection pruning is central to most pruning algorithms. The program analyzes the individual connections between the neurons to determine which connections have the least impact on the effectiveness of the neural network. Connections are not the only thing that the program can prune. Analyzing the pruned connections will reveal that the program can also prune individual neurons.

11.1.2 Pruning Neurons

Pruning focuses primarily on the connections between the individual neurons of the neural network. However, to prune individual neurons, we must examine the connections between each neuron and the other neurons. 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, neurons that have no connections are the end result because the program has pruned all of the neuron's connections. Then the program can prune this type of a neuron.

11.1.3 Improving or Degrading Performance

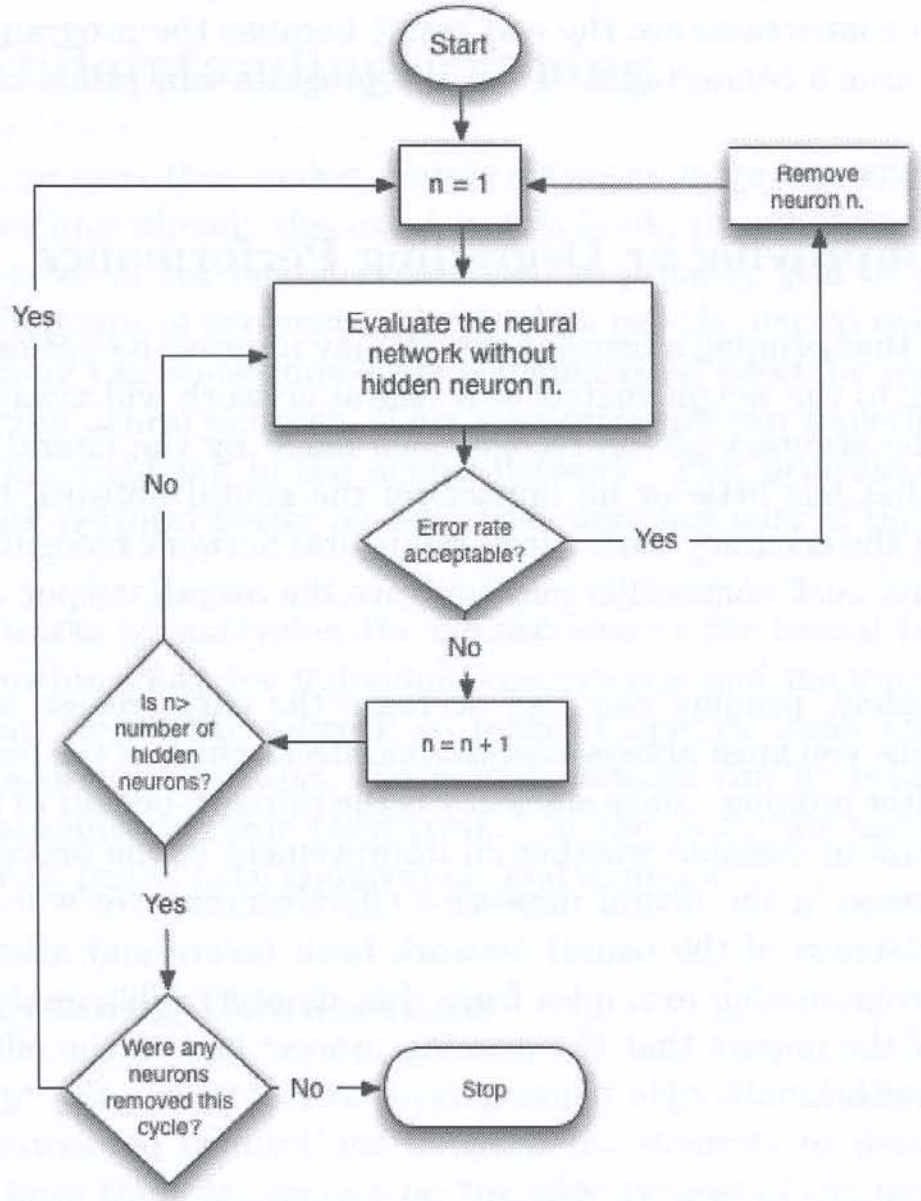
It is possible that pruning a neural network may improve its performance. Any modifications to the weight matrix of a neural network will always have some impact on the accuracy of the recognitions made by the neural network. A connection that has little or no impact on the neural network may actually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

Unfortunately, pruning can also decrease the effectiveness of the neural network. Thus, you must always 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. We will evaluate the overall effectiveness of the neural network both before and after pruning in one of the programming examples from this chapter. This analysis will give us an idea of the impact that the pruning process has on the effectiveness of the neural network.

11.2 Pruning Algorithm

We will now review exactly how pruning takes place. Pruning works by examining the weight matrices of a previously trained neural network. The pruning algorithm will then attempt to remove neurons without disrupting the output of the neural network. Figure 11.1 shows the algorithm used for selective pruning:

Figure 11.1: Pruning a Neural Network


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

As you can see, the pruning algorithm has a trial-and-error approach. The pruning algorithm attempts to remove neurons from the neural network until it cannot remove additional neurons 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 program evaluates the error level of the neural network both with and without the specified neuron. If the error rate jumps beyond a predefined level, the program retains the neuron and evaluates the next. If the error rate does not improve significantly, the program removes the neuron.

Once the program has evaluated all neurons, it 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 it has fewer hidden neurons.

11.3 Model Selection

Model selection is the process where the programmer attempts to find a set of hyper-parameters that produce the best neural network, or other machine learning model. In this book, we have mentioned many different hyper-parameters that are the settings that you must provide to the neural network framework. Examples of neural network hyper-parameters include:

- The number of hidden layers
- The order of the convolutional, pooling, and dropout layers
- The type of activation function
- The number of hidden neurons
- The structure of pooling and convolutional layers

As you've read through these chapters that mention hyper-parameters, you've probably been wondering how you know which settings to use. Unfortunately, there is no easy answer. If easy methods existed that determine these settings, programmers would have constructed the neural network frameworks that automatically set these hyper-parameters for you.

While we will provide more insight into hyper-parameters in Chapter 14, "Architecting Neural Networks," you will still need to use the model selection processes described in this chapter. Unfortunately, model selection is very time-consuming. We spent 90% of our time performing model selection during our last Kaggle competition. Often, success in modeling is closely related to the amount of time you have to spend on model selection.

11.3.1 Grid Search Model Selection

Grid search is a trial-and-error, brute-force algorithm. For this technique, you must specify every combination of the hyper-parameters that you would like to use. You must be judicious in your selection because the number of search iterations can quickly grow. Typically, you must specify the hyper-parameters that you would like to search. This specification might look like the following:

- Hidden Neurons: 2 to 10, step size 2
- Activation Functions: tanh, sigmoid & ReLU

The first item states that the grid search should try hidden neuron counts between 2 and 10 counting by 2, which results in the following: 2, 4, 6, 8, and 10 (5 total possibilities.) The second item states that we should also try the activation functions tanh, sigmoid, and ReLU for each neuron count. This process results in a total of fifteen iterations because five possibilities times three possibilities is fifteen total. These possibilities are listed here:

```
Iteration #1: [2][sigmoid]
Iteration #2: [4][sigmoid]
Iteration #3: [6][sigmoid]
Iteration #4: [8][sigmoid]
Iteration #5: [10][sigmoid]
Iteration #6: [2][ReLU]
Iteration #7: [4][ReLU]
Iteration #8:  [6][ReLU]
Iteration #9:  [8][ReLU]
Iteration #10: [10][ReLU]
Iteration #11: [2][tanh]
Iteration #12: [4][tanh]
Iteration #13: [6][tanh]
Iteration #14: [8][tanh]
Iteration #15: [10][tanh]
```

Each set of possibilities is called an axis. These axes rotate through all possible combinations before they finish. You can visualize this process by thinking of a car's odometer. The far left dial (or axis) is spinning the fastest. It counts between 0 and 9. Once it hits 9 and needs to go to the next number, it forward back to 0, and the next place, to the left, rolls forward by one. Unless that next place was also on 9, the rollover continues to the left. At some point, all digits on the odometer are at 9, and the entire device would roll back over to 0. When this final rollover occurs, the grid search is done.

Most frameworks allow two axis types. The first type is a numeric range with a step. The second type is a list of values, like the activation functions above. The following Javascript example allows you to try your own sets of axes to see the number of iterations produced:

http://www.heatonresearch.com/aifh/vol3/grid_iter.html

Listing 11.1 shows the pseudocode necessary to roll through all iterations of several sets of values:

Listing 11.1: Grid Search

```
# The variable axes contains a list of each axis.
# Each axes (in axes) is a list of possible values
# for that axis.
# Current index of each axis is zero, create an array
# of zeros.
indexes = zeros(len(axes))
done = false
while not done:
# Prepare vector of current iterationŠs
# hyper-parameters.
  iteration = []
  for i from 0 to len(axes)
    iteration.add(axes [i][indexes[i]] )
# Perform one iteration , passing in the hyper-parameters
# that are stored in the iteration list. This function
# should train the neural network according to the
# hyper-parameters and keep note of the best trained
# network so far.
  perform_iteration(iteration)
# Rotate the axes forward one unit , like a carŠs
# odometer.
  indexes[0] = indexes[0] + 1;
  var counterIdx = 0;
# roll forward the other places , if needed
  while not done and indexes[counterIdx]>=
        len(axes [counterIdx]):
    indexes[counterIdx] = 0
    counterIdx = counterIdx + 1
    if counterIdx>=len(axes):
      done = true
    else:
      indexes[counterIdx] = indexes[counterIdx] + 1
```

The code above uses two loops to pass through every possible set of the hyper-parameters. The first loop continues while the program is still producing hyper-parameters. Each time through, this loop increases the first hyper-parameter to the next value. The second loop detects if the first hyper-parameter has rolled over. The inner loop keeps moving forward to the next hyper-parameter until no more rollovers occur. Once all the hyper-parameters roll over, the process is done.

As you can see, the grid search can quickly result in a large number of iterations. Consider if you wished to search for the optimal number of hidden neurons on five layers, where you allowed up to 200 neurons on each level. This value would be equal to 200 multiplied by itself five times, or 200 to the fifth power. This process results in 320 billion iterations. Because each iteration involves training a neural network, iterations can take minutes, hours or even days to execute.

When performing grid searches, multi-threading and grid processing can be beneficial. Running the iterations through a thread pool can greatly speed up the search. The thread pool should have a size equal to the number of cores on the computer's machine. This trait allows a machine with eight cores to work on eight neural networks simultaneously. The training of the individual models must be single threaded when you run the iterations simultaneously. Many frameworks will use all available cores to train a single neural network. When you have a large number of neural networks to train, you should always train several neural networks in parallel, running them one a time so that each network uses the machines cores.

11.3.2 Random Search Model Selection

It is also possible to use a random search for model selection. Instead of systematically trying every hyper-parameter combination, the random search method chooses random values for hyper-parameters. For numeric ranges, you no longer need to specify a step value, the random model selection will choose a continuous range of floating point numbers between your specified beginning and ending points. For a random search, the programmer typically specifies either a time or an iteration limit. The following shows a random search, using the same axes as above, but it is limited to ten iterations:

```
Iteration #1: [3.298266736790538][sigmoid]
Iteration #2: [9.569985574809834][ReLU]
Iteration #3: [1.241154231596738][sigmoid]
Iteration #4: [9.140498645836487][sigmoid]
Iteration #5: [8.041758658131585][tanh]
Iteration #6: [2.363519841339439][ReLU]
Iteration #7: [9.72388393455185][tanh]
Iteration #8: [3.411276006139815][tanh]
Iteration #9: [3.1166220877785236][sigmoid]
Iteration #10: [8.559433702612296][sigmoid]
```

As you can see, the first axis, which is the hidden neuron count, is now taking on floating-point values. You can solve this problem by rounding the neuron count to the nearest whole number. It is also advisable to avoid retesting the same hyper-parameters more than once. As a result, the program should keep a list of previously tried hyper-parameters so that it doesn't repeat any hyper-parameters that were with a small range of a previously tried set.

The following URL uses Javascript to show random search in action:

http://www.heatonresearch.com/aifh/vol3/random_iter.html

11.3.3 Other Model Selection Techniques

Model selection is a very active area of research, and, as a result, many innovative ways exist to perform it. Think of the hyper-parameters as a vector of values and the process of finding the best neural network score for those hyper-parameters as an objective function. You can consider these hyper-parameters as an optimization problem. We have previously examined many optimization algorithms in earlier volumes of this book series. These algorithms are the following:

- Ant Colony Optimization (ACO)
- Genetic Algorithms
- Genetic Programming
- Hill Climbing
- Nelder-Mead
- Particle Swarm Optimization (PSO)
- Simulated Annealing

We examined many of these algorithms in detail in Volumes 1 and 2 of *Artificial Intelligence for Humans.* Although the list of algorithms is long, the reality is that most of these algorithms are not suited for model selection because the objective function for model selection is computationally expensive. It might take minutes, hours or even days to train a neural network and determine how well a given set of hyper-parameters can train a neural network.

Nelder-Mead and sometimes hill climbing turn out to be the best options if you wish to apply an optimization function to model selection. These algorithms attempt to minimize calls to the objective function. Calls to the objective function are very expensive for a parameter search because a neural network must be trained. A good technique for optimization is to generate a set of hyper-parameters to use as a starting point for Nelder-Mead and allow Nelder-Mead to improve these hyper-parameters. Nelder-Mead is a good choice for a hyper-parameter search because it results in a relatively small number of calls to the objective function.

Model selection is a very common part of Kaggle data science competitions. Based on competition discussions and reports, most participants use grid and random searches for model selection.. Nelder-Mead is also popular. Another technique that is gaining in popularity is the use of Bayesian optimization, as described by Snoek, Larochelle, Hugo & Adams (2012). An implementation of this algorithm, written in Python, is called Spearmint, and you can find it at the following URL:

https://github.com/JasperSnoek/spearmint

Bayesian optimization is a relatively new technique for model selection on which we have only recently conducted research. Therefore, this current book does not contain a more profound examination of it. Future editions may include more information of this technique.

11.4 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. Execution speed, number of connections, and error are all measures of efficiency. Although neural networks must be effective at recognizing patterns, efficiency is the main goal of pruning. Several different algorithms can prune a neural network. In this chapter, we examined two of these algorithms. If your neural network is already operating sufficiently fast, you must evaluate whether the pruning is justified. Even when efficiency is important, you must weigh the trade-offs between efficiency and a reduction in the effectiveness of your neural network.

Model selection plays a significant role in neural network development. Hyper-parameters are settings such as hidden neuron, layer count, and activation function selection. Model selection is the process of finding the set of hyper-parameters that will produce the best-trained neural network. A variety of algorithms can search through the possible settings of the hyper-parameters and find the best set.

Pruning can sometimes lead to a decrease in the tendency for neural networks to overfit. This reduction in overfitting is typically only a byproduct of the pruning process. Algorithms that reduce overfitting are called regularization algorithms. Although pruning will sometimes have a regularizing effect, an entire group of algorithms, called regularization algorithms, exist to reduce overfitting. We will focus exclusively on these algorithms in the next chapter.

[illegible] of identifying [illegible] through [illegible] as many of the items [illegible] and then the task set.

Sorting can sometimes happen [illegible] the same as list searching [illegible] works correctly. This [illegible] chapter [illegible] the previous [illegible]. Algorithms that [illegible] are called [illegible] two algorithms, although [illegible] have [illegible] as [illegible] called [illegible] algorithms [illegible] We will [illegible]