Chapter 12 Dropout and Regularization

- Regularization
- L1 & L2 Regularization
- Dropout Layers

Regularization is a technique that reduces overfitting, which occurs when neural networks attempt to memorize training data, rather than learn from it. Humans are capable of overfitting as well. Before we examine the ways that a machine accidentally overfits, we will first explore how humans can suffer from it.

Human programmers often take certification exams to show their competence in a given programming language. To help prepare for these exams, the test makers often make practice exams available. Consider a programmer who enters a loop of taking the practice exam, studying more, and then taking the practice exam again. At some point, the programmer has memorized much of the practice exam, rather than learning the techniques necessary to figure out the individual questions. The programmer has now overfit to the practice exam. When this programmer takes the real exam, his actual score will likely be lower than what he earned on the practice exam.

A computer can overfit as well. Although a neural network received a high score on its training data, this result does not mean that the same neural network will score high on data that was not inside the training set. Regularization is one of the techniques that can prevent overfitting. A number of different regularization techniques exist. Most work by analyzing and potentially modifying the weights of a neural network as it trains.

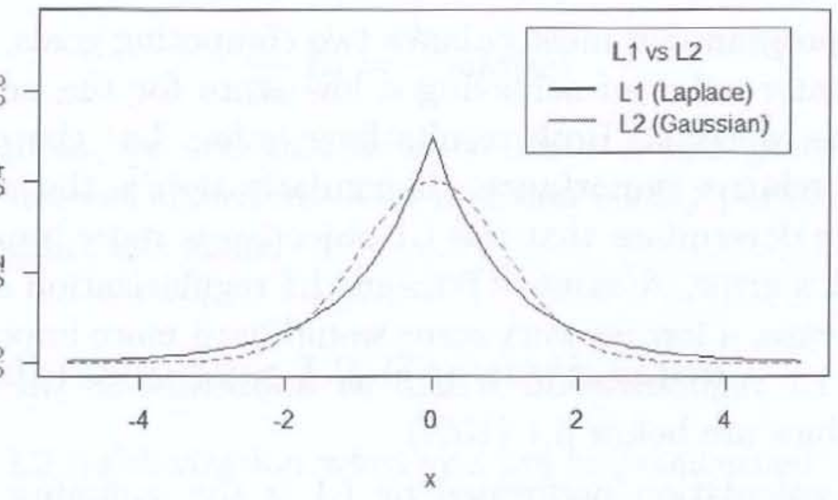
12.1 L1 and L2 Regularization

L1 and L2 regularization are two common regularization techniques that can reduce the effects of overfitting (Ng, 2004). Both of these algorithms can either work with an objective function or as a part of the backpropagation algorithm. In both cases the regularization algorithm is attached to the training algorithm by adding an additional objective.

Both of these algorithms work by adding a weight penalty to the neural network training. This penalty encourages the neural network to keep the weights to small values. Both L1 and L2 calculate this penalty differently. For gradient-descent-based algorithms, such as backpropagation, you can add this penalty calculation to the calculated gradients. For objective-function-based training, such as simulated annealing, the penalty is negatively combined with the objective score.

Both L1 and L2 work differently in the way that they penalize the size of a weight. L1 will force the weights into a pattern similar to a Gaussian distribution; the L2 will force the weights into a pattern similar to a Laplace distribution, as demonstrated by Figure 12.1:

Figure 12.1: L1 vs L2



As you can see, L1 algorithm is more tolerant of weights further from 0, whereas the L2 algorithm is less tolerant. We will highlight other important differences between L1 and L2 in the following sections. You also need to note that both L1 and L2 count their penalties based only on weights; they do not count penalties on bias values.

12.1.1 Understanding L1 Regularization

You should use L1 regularization to create sparsity in the neural network. In other words, the L1 algorithm will push many weight connections to near 0. When a weight is near 0, the program drops it from the network. Dropping weighted connections will create a sparse neural network.

Feature selection is a useful byproduct of sparse neural networks. Features are the values that the training set provides to the input neurons. Once all the weights of an input neuron reach 0, the neural network training determines that the feature is unnecessary. If your data set has a large number of input features that may not be needed, L1 regularization can help the neural network detect and ignore unnecessary features.

Equation 12.1 shows the penalty calculation performed by L1:

$$E_1 = \lambda_1 \sum_w |w| \tag{12.1}$$

Essentially, a programmer must balance two competing goals. He or she must decide the greater value of achieving a low score for the neural network or regularizing the weights. Both results have value, but the programmer has to choose the relative importance. If regularization is the main goal, the λ (lambda) value determines that the L1 objective is more important than the neural network's error. A value of 0 means L1 regularization is not considered at all. In this case, a low network error would have more importance. A value of 0.5 means L1 regularization is half as important as the error objective. Typical L1 values are below 0.1 (10%).

The main calculation performed by L1 is the summing of the absolute values (as indicated by the vertical bars) of all the weights. The bias values are not summed.

If you are using an optimization algorithm, such as simulated annealing, you can simply combine the value returned by Equation 12.1 to the score. You should combine this value to the score in such a way so that it has a negative effect. If you are trying to minimize the score, then you should add the L1 value. Similarly, if you are trying to maximize the score, then you should subtract the L1 value.

If you are using L1 regularization with a gradient-descent-based training algorithm, such as backpropagation, you need to use a slightly different error term, as shown by Equation 12.2:

$$E_1 = \frac{\lambda_1}{n} \sum_w |w| \tag{12.2}$$

Equation 12.2 is nearly the same as Equation 12.1 except that we divide by n. The value n represents the number of training set evaluations. For example, if there were 100 training set elements and three output neurons, n would be 300. We derive this number because the program has three values to evaluate for each of those 100 elements. It is necessary to divide by n because the program applies Equation 12.2 at every training evaluation. This characteristic contrasts with Equation 12.1, which is applied once per training iteration.

To use Equation 12.2, we need to take its partial derivative with respect to the weight. Equation 12.3 shows the partial derivative of Equation 12.2:

$$\frac{\partial}{\partial w} E_1 = \frac{\lambda_1}{n} sgn(w) \tag{12.3}$$

To use this gradient, we add this value to every weight gradient calculated by the gradient-descent algorithm. This addition is only performed for weight values; the biases are left alone.

12.1.2 Understanding L2 Regularization

You should use L2 regularization when you are less concerned about creating a space network and are more concerned about low weight values. The lower weight values will typically lead to less overfitting.

Equation 12.4 shows the penalty calculation performed by L2:

$$E_2 = \lambda_2 \sum_w w^2 \tag{12.4}$$

Like the L1 algorithm, the λ (lambda) value determines how important the L2 objective is compared to the neural network's error. Typical L2 values are below 0.1 (10%). The main calculation performed by L2 is the summing of the squares of all of the weights. The bias values are not summed.

If you are using an optimization algorithm, such as simulated annealing, you can simply combine the value returned by Equation 12.4 to the score. You should combine this value with the score in such a way so that it has a negative effect. If you are trying to minimize the score, then you should add the L2 value. Similarly, if you are trying to maximize the score, then you should subtract the L2 value.

If you are using L2 regularization with a gradient-descent-based training algorithm, such as backpropagation, you need to use a slightly different error term, as shown by Equation 12.5:

$$E_2 = \frac{\lambda_2}{n} \sum_w w^2 \tag{12.5}$$

Equation 12.5 is nearly the same as Equation 12.4, except that, unlike L1, we take the squares of the weights. To use Equation 12.5, we need to take the partial derivative with respect to the weight. Equation 12.6 shows the partial derivative of Equation 12.6:

$$\frac{\partial}{\partial w} E_2 = \frac{\lambda_2}{n} w \tag{12.6}$$

To use this gradient, you need to add this value to every weight gradient calculated by the gradient-descent algorithm. This addition is only performed on weight values; the biases are left alone.

12.2 Dropout Layers

Hinton, Srivastava, Krizhevsky, Sutskever, & Salakhutdinov (2012) introduced the dropout regularization algorithm. Although dropout works in a different way than L1 and L2, it accomplishes the same goal–the prevention of overfitting. However, the algorithm goes about the task by actually removing neurons and connections–at least temporarily. Unlike L1 and L2, no weight penalty is added. Dropout does not directly seek to train small weights.

Dropout works by causing hidden neurons of the neural network to be unavailable during part of the training. Dropping part of the neural network causes the remaining portion to be trained to still achieve a good score even without the dropped neurons. This decreases coadaption between neurons, which results in less overfitting.

12.2.1 Dropout Layer

Most neural network frameworks implement dropout as a separate layer. Dropout layers function as a regular, densely connected neural network layer. The only difference is that the dropout layers will periodically drop some of their neurons during training. You can use dropout layers on regular feedforward neural networks. In fact, they can also become layers in convolutional LeNET-5 networks like we studied in Chapter 10, "Convolutional Neural Networks."

The usual hyper-parameters for a dropout layer are the following:

- Neuron Count
- Activation Function
- Dropout Probability

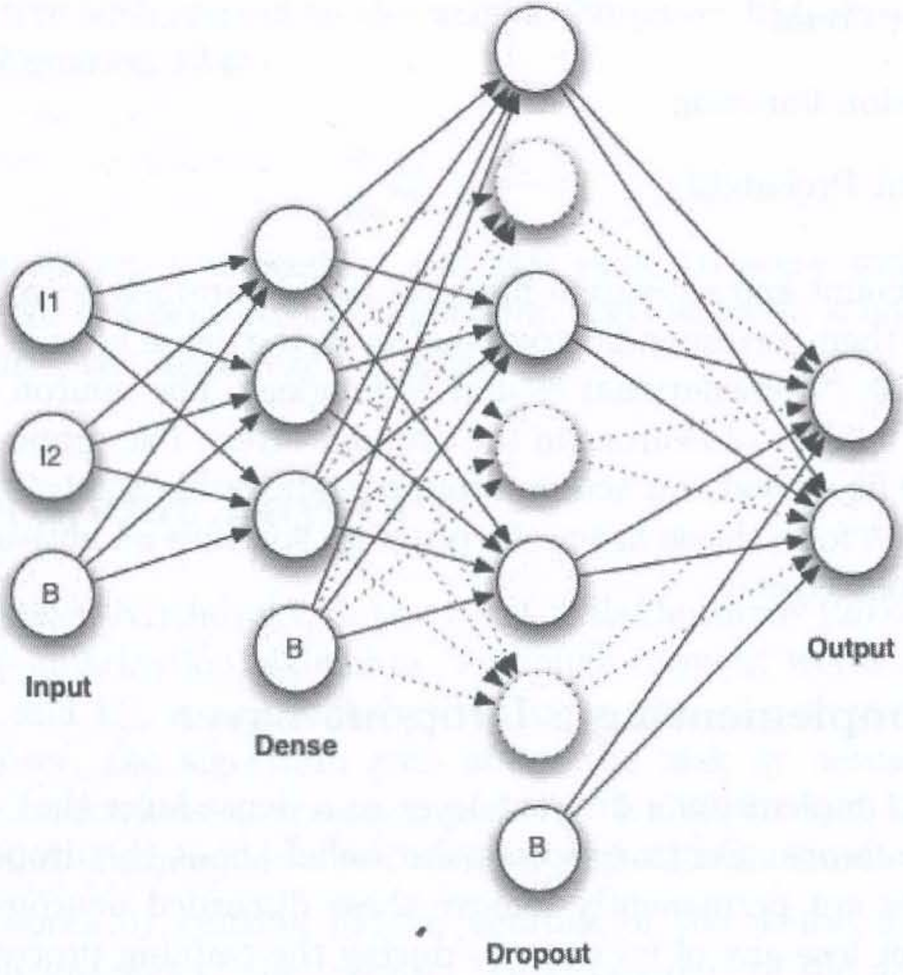
The neuron count and activation function hyper-parameters work exactly the same way as their corresponding parameters in the dense layer type mentioned in Chapter 10, "Convolutional Neural Networks." The neuron count simply specifies the number of neurons in the dropout layer. The dropout probability indicates the likelihood of a neuron dropping out during the training iteration. Just as it does for a dense layer, the program specifies an activation function for the dropout layer.

12.2.2 Implementing a Dropout Layer

The program implements a dropout layer as a dense layer that can eliminate some of its neurons. Contrary to popular belief about the dropout layer, the program does not permanently remove these discarded neurons. A dropout layer does not lose any of its neurons during the training process, and it will still have exactly the same number of neurons after training. In this way, the program only temporarily masks the neurons rather than dropping them.

Figure 12.2 shows how a dropout layer might be situated with other layers:

Figure 12.2: Dropout Layer



The discarded neurons and their connections are shown as dashed lines. The input layer has two input neurons as well as a bias neuron. The second layer is a dense layer with three neurons as well as a bias neuron. The third layer is a dropout layer with six regular neurons even though the program has dropped 50% of them. While the program drops these neurons, it neither calculates nor trains them. However, the final neural network will use all of these neurons for the output. As previously mentioned, the program only temporarily discards the neurons.

During subsequent training iterations, the program chooses different sets of neurons from the dropout layer. Although we chose a probability of 50% for dropout, the computer will not necessarily drop three neurons. It is as if we flipped a coin for each of the dropout candidate neurons to choose if that neuron was dropped out. You must know that the program should never drop the bias neuron. Only the regular neurons on a dropout layer are candidates.

The implementation of the training algorithm influences the process of discarding neurons. The dropout set frequently changes once per training iteration or batch. The program can also provide intervals where all neurons are present. Some neural network frameworks give additional hyper-parameters to allow you to specify exactly the rate of this interval.

Why dropout is capable of decreasing overfitting is a common question. The answer is that dropout can reduce the chance of a codependency developing between two neurons. Two neurons that develop a codependency will not be able to operate effectively when one is dropped out. As a result, the neural network can no longer rely on the presence of every neuron, and it trains accordingly. This characteristic decreases its ability to memorize the information presented to it, thereby forcing generalization.

Dropout also decreases overfitting by forcing a bootstrapping process upon the neural network. Bootstrapping is a very common ensemble technique. We will discuss ensembling in greater detail in Chapter 16, "Modeling with Neural Networks." Basically, ensembling is a technique of machine learning that combines multiple models to produce a better result than those achieved by individual models. Ensemble is a term that originates from the musical ensembles in which the final music product that the audience hears is the combination of many instruments.

Bootstrapping is one of the most simple ensemble techniques. The programmer using bootstrapping simply trains a number of neural networks to perform exactly the same task. However, each of these neural networks will perform differently because of some training techniques and the random numbers used in the neural network weight initialization. The difference in weights causes the performance variance. The output from this ensemble of neural networks becomes the average output of the members taken together. This process decreases overfitting through the consensus of differently trained neural networks.

Dropout works somewhat like bootstrapping. You might think of each neural network that results from a different set of neurons being dropped out as an individual member in an ensemble. As training progresses, the program creates more neural networks in this way. However, dropout does not require the same amount of processing as does bootstrapping. The new neural networks created are temporary; they exist only for a training iteration. The final result is also a single neural network, rather than an ensemble of neural networks to be averaged together.

12.3 Using Dropout

In this chapter, we will continue to evolve the book's MNIST handwritten digits example. We examined this data set in the book introduction and used it in several examples.

The example for this chapter uses the training set to fit a dropout neural network. The program subsequently evaluates the test set on this trained network to view the results. Both dropout and non-dropout versions of the neural network have results to examine.

The dropout neural network used the following hyper-parameters:

- Activation Function: ReLU
- Input Layer: 784 (28x28)
- Hidden Layer 1: 1000

- Dropout Layer: 500 units, 50%
- Hidden Layer 2: 250
- Output Layer: 10 (because there are 10 digits)

We selected the above hyper-parameters through experimentation. By rounding the number of input neurons up to the next even unit, we chose a first hidden layer of 1000. The next three layers constrained this amount by half each time. Placing the dropout layer between the two hidden layers provided the best improvement in the error rate. We also tried placing it both before hidden layer 1 and after hidden layer 2. Most of the overfitting occurred between the two hidden layers.

We used the following hyper-parameters for the regular neural network. This process is essentially the same as the dropout network except that an additional hidden layer replaces the dropout layer.

- Activation Function: ReLU
- Input Layer: 784 (28x28)
- Hidden Layer 1: 1000
- Hidden Layer 2: 500
- Hidden Layer 3: 250
- Output Layer: 10 (because there are 10 digits)

The results are shown here:

```
Relu:
Best valid loss was 0.068229 at epoch 17.
Incorrect 170/10000 (1.7000000000000002%)
ReLU+Dropout:
Best valid loss was 0.065753 at epoch 5.
Incorrect 120/10000 (1.2%)
```

As you can see, dropout neural network achieved a better error rate than the ReLU only neural network from earlier in the book. By reducing the amount of overfitting, the dropout network got a better score. You should also notice that, although the non-dropout network did achieve a better training score, this result is not good. It indicates overfitting. Of course, these results will vary, depending on the platform used.

12.4 Chapter Summary

We introduced several regularization techniques that can reduce overfitting. When the neural network memorizes the input and expected output, overfitting occurs because the program has not learned to generalize. Many different regularization techniques can force the neural network to learn to generalize. We examined L1, L2, and dropout. L1 and L2 work similarly by imposing penalties for weights that are too large. The purpose of these penalties is to reduce complexity in the neural network. Dropout takes an entirely different approach by randomly removing various neurons and forcing the training to continue with a partial neural network.

The L1 algorithm penalizes large weights and forces many of the weights to approach 0. We consider the weights that contain a zero value to be dropped from the neural network. This reduction produces a sparse neural network. If all weighted connections between an input neuron and the next layer are removed, you can assume that the feature connected to that input neuron is unimportant. Feature selection is choosing input features based on their importance to the neural network. The L2 algorithm penalizes large weights, but it does not tend to produce neural networks that are as sparse as those produced by the L1 algorithm.

Dropout randomly drops neurons in a designated dropout layer. The neurons that were dropped from the network are not gone as they were in pruning. Instead, the dropped neurons are temporarily masked from the neural network. The set of dropped neurons changes during each training iteration. Dropout forces the neural network to continue functioning when neurons are removed. This makes it difficult for the neural network to memorize and overfit.

So far, we have explored only feedforward neural networks in this volume.

In this type of network, the connections only move forward from the input layer to hidden layers and ultimately to the output layer. Recurrent neural networks allow backward connections to previous layers. We will analyze this type of neural network in the next chapter.

Additionally, we have focused primarily on using neural networks to recognize patterns. We can also teach neural networks to predict future trends. By providing a neural network with a series of time-based values, it can predict subsequent values. In the next chapter, we will also demonstrate predictive neural networks. We refer to this type of neural network as a temporal neural network. Recurrent neural networks can often make temporal predictions.