THE DATA SCIENCE LAB

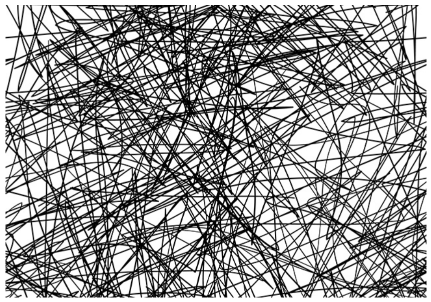
Neural Network L1 Regularization Using Python

The data science doctor continues his exploration of techniques used to reduce the likelihood of model overfitting, caused by training a neural network for too many iterations.

By James McCaffrey 12/05/2017

GET CODE DOWNLOAD

MOST POPULAR



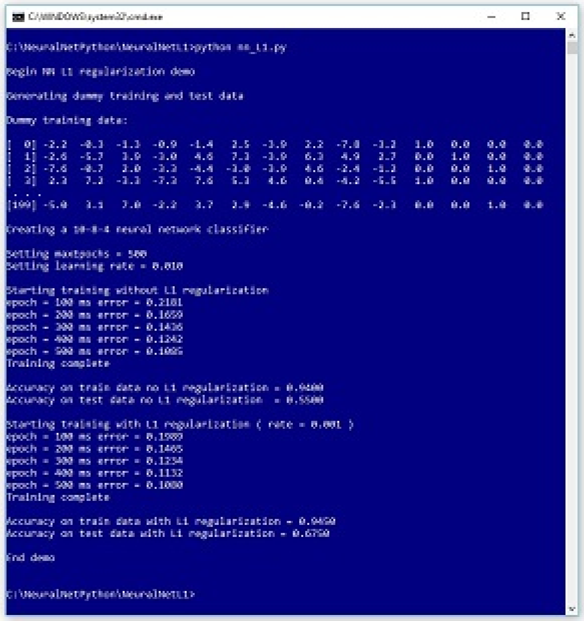
Regularization is a technique used to reduce the likelihood of neural network model overfitting. Model overfitting can occur when you train a neural network for too many iterations. This sometimes results in a situation where the trained neural network model predicts the output values for the training data very well, with little error and high accuracy, but when the trained model is applied to new, previously unseen data, the model predicts poorly.

There are several forms of regularization. The two most common forms are called L1 and L2 regularization. This article focuses on L1 regularization, but I'll discuss L2 regularization briefly.

You can think of a neural network as a complex math function that makes predictions. Training is the process of finding values for the network weights and bias constants that effectively define the behavior of the network. The most common way to train a neural network is to use a set of training data with known input values and known, correct output values. You apply an optimization algorithm, typically back-propagation, to find weights and bias values that minimize some error metric between the computed output values and the correct output values (typically squared error or cross entropy error).

Overfitting is often characterized by weights that have large magnitudes, such as +38.5087 and -23.182, rather than small magnitudes such as +1.650 and -3.043. L1 regularization reduces the possibility of overfitting by keeping the values of the weights and biases small.

A good way to see where this article is headed is to take a look at the screenshot of a demo program in **Figure 1**. The demo program is coded using raw Python (no libraries) with the NumPy numeric library, but you should have no trouble refactoring to another language, such as C# or Visual Basic, if you want.



```
C:\NeuralNetPython\NeuralNetL1>python nn_L1.py

Begin NN L1 regularization demo

Generating dummy training and test data

Dummy training data:

[  0] -2.2  -0.3  -1.3  -0.9  -1.4   2.5  -3.9   2.2  -7.8  -3.2   1.0   0.0   0.0   0.0
[  1] -2.6  -5.7   3.9  -3.0   4.6   7.3  -3.9   6.3   4.9   2.7   0.0   1.0   0.0   0.0
[  2] -7.6  -0.7   2.0  -3.2  -4.4  -1.0  -3.9   4.6  -2.4  -1.2   0.0   0.0   1.0   0.0
[  3]  2.3   7.2  -3.3  -7.2   7.6   5.3   4.6   0.6  -4.2  -5.5   1.0   0.0   0.0   0.0
 . . .
[199] -5.0   3.1   7.0  -2.2   3.7   3.9  -4.6  -0.2  -7.6  -2.3   0.0   0.0   1.0   0.0

Creating a 10-8-4 neural network classifier

Setting maxEpochs = 500
Setting learning rate = 0.010

Starting training without L1 regularization
epoch = 100 ms error = 0.2181
epoch = 200 ms error = 0.1659
epoch = 300 ms error = 0.1436
epoch = 400 ms error = 0.1242
epoch = 500 ms error = 0.1085
Training complete

Accuracy on train data no L1 regularization = 0.9400
Accuracy on test data no L1 regularization  = 0.5500

Starting training with L1 regularization ( rate = 0.001 )
epoch = 100 ms error = 0.1989
epoch = 200 ms error = 0.1465
epoch = 300 ms error = 0.1234
epoch = 400 ms error = 0.1132
epoch = 500 ms error = 0.1080
Training complete

Accuracy on train data with L1 regularization = 0.9450
Accuracy on test data with L1 regularization = 0.6750

End demo


C:\NeuralNetPython\NeuralNetL1>
```

[Click on image for larger view.]
Neural Network L1 Regularization Demo Program

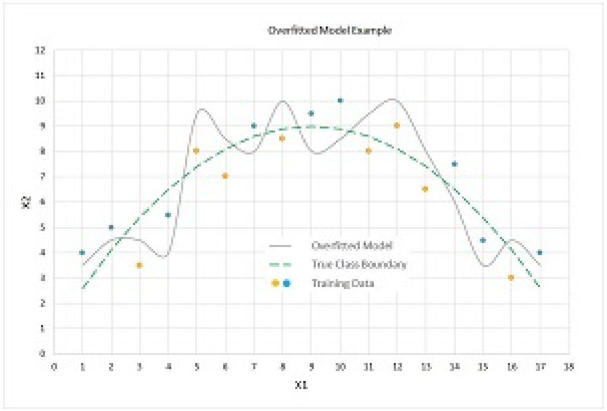
The demo begins by using a utility neural network to generate 200 synthetic training items and 40 test items. Each data item has 10 input predictor variables (often called features) and four output variables (often called class labels) that represent 1-of-N encoded categorical data. For example, if you were trying to predict the political leaning of a person, and there are just four possible leanings, you could encode conservative as (1, 0, 0, 0), moderate as (0, 1, 0, 0), liberal as (0, 0, 1, 0) and radical as (0, 0, 0, 1).

The demo program creates a neural network classifier with 10 input nodes, eight hidden processing nodes and four output nodes. The number of input and output nodes is determined by the data, but the number of hidden nodes is a free parameter and must be determined by trial and error. The demo program trains a first model using the back-propagation algorithm without L1 regularization. That first model gives 94.00 percent accuracy on the training data (188 of 200 correct) and a poor 55.00 percent accuracy on the test data (just 22 of 40 correct). It appears that the model might be overfitted. To be honest, in order to get a demo result that illustrates typical regularization behavior, I cheated a bit by setting the number of training iterations to a small value (500), and the model is actually underfitted -- not trained enough.

The demo continues by training a second model, this time with L1 regularization. The second model gives 94.50 percent accuracy on the training data (189 of 200 correct) and 67.50 percent accuracy on the test data (27 of 40 correct). In this example, using L1 regularization has made a significant improvement in classification accuracy on the test data.

Understanding Neural Network Model Overfitting

Model overfitting is often a significant problem when training a neural network. The idea is illustrated in the graph in **Figure 2**. There are two predictor variables: X1 and X2. There are two possible categorical classes, indicated by the orange (class = 0) and blue (class = 1) dots. You can imagine this corresponds to the problem of predicting if a loan application should be approved (1) or denied (0), based on normalized income (X1) and normalized debt (X2). For example, the left-most data point at (X1 = 1, X2 = 4) is colored blue (approve).



[Click on image for larger view.]
Neural Network Overfitting Example

The dashed green line represents the actual boundary between the two classes. This boundary is unknown to you. Notice that data items that are above the green line are mostly blue (7 of 9 points) and that data items below the green line are mostly orange (6 of 8 points). The four misclassifications in the training data are due to randomness inherent in almost all real-life data.

A good neural network model would find the true decision boundary represented by the dashed green line. However, if you train a neural network model too long, it will essentially get too good and produce a model indicated by the solid wavy gray line. Notice that the gray line makes perfect predictions on the test data: All the blue dots are above the gray line and all the orange dots are below the gray line.

However, when the overfitted model is presented with new, previously unseen data, there's a good chance the model will make an incorrect prediction. For example, a new data item at (X1 = 11, X2 = 9) is above the green dashed truth boundary and so should be classified as blue. But because the data item is below the gray line overfitted boundary, it will be incorrectly classified as orange.

If you vaguely remember your high school algebra you might recall that the overfitted gray line, with its peaky shape, looks like the graph of a polynomial function that has coefficients with large magnitudes. These coefficients correspond to neural network weights and biases. Therefore, the idea behind L1 regularization is to keep the magnitudes of the weights and bias values small, which will prevent a spikey decision boundary, which in turn will avoid model overfitting.

Understanding L1 Regularization

In a nutshell, L1 regularization works by adding a term to the error function used by the training algorithm. The additional term penalizes large-magnitude weight values. By far the two most common error functions used in neural network training are squared error and cross entropy error. For the rest of this article I'll assume squared error, but the ideas are exactly the same when using cross entropy error.

For L1 regularization, the weight penalty term that's added to the error term is a small fraction (often given the Greek letter lowercase lambda) of the sum of the absolute values of the weights. For example, suppose you have a neural network with only three weights. If you set lambda = 0.10 (actual values of the L1 constant are usually much smaller), and if the current values of the weights are (6.0, -2.0, 4.0) then in addition to the base squared error between computed output values and correct target output values, the augmented error term adds 0.10 * [abs(6.0) + abs(-2.0) + abs(4.0)] = 0.10 * (6.0 + 2.0 + 4.0) = 0.10 * 12.0 = 1.20 to the overall error.

The key math equations (somewhat simplified for clarity) are shown in **Figure 3**. The bottom-most equation is the weight update for a weight connecting a hidden node j to an output node k. In words, "The new weight value is the old weight value plus a delta value."

$$E = \underbrace{\frac{1}{2} * \sum (t_k - o_k)^2}_{\text{squared error}} + \underbrace{\lambda * \sum |w_i|}_{\text{L1 weight penalty}}$$

$$\Delta w_{jk} = -1 * \underset{\substack{\uparrow \\ \text{learning rate}}}{\eta} * \overbrace{[\, x_j * \underbrace{(o_k - t_k) * o_k * (1 - o_k) \,] \pm \lambda \,]}_{\text{signal}}}^{\frac{\partial E}{\partial w_{jk}} \ \text{gradient}}$$

$$w_{jk} = w_{jk} + \Delta w_{jk}$$

[Click on image for larger view.]
L1 Regularization Equations

An example line of code in the demo is:

```
self.hoWeights[j,k] += delta
```

The top-most equation is a squared error with an L1 penalty term. In words, "Take each target value (tk), subtract the computed output value (ok), square the difference, add all the sums, and divide by 2, then add a small constant lambda times the sum of the absolute values of the weights."

The middle equation shows how a weight delta is calculated, assuming a squared error with L1 weight penalty. Overall, a delta is -1 times a small learning rate constant (Greek eta, which looks like lowercase script "n") times the gradient of the error function. The gradient is the Calculus derivative of the error function. The error function has two parts, the basic error plus the weight penalty. The derivative of a sum is the sum of the derivatives. The derivative of the left part of the error term is quite tricky and outside the scope of this article, but you can see it uses target values, computed output values and hidden node values (the xj).

To use L1 regularization, you need the Calculus derivative of weight penalty as part of the error term. As it turns out, if a weight is positive, then the derivative of the absolute value is just the constant +1 times lambda. If a weight is negative, the derivative is -1 times lambda. The absolute value function doesn't have a derivative at 0, but this isn't a problem. If a weight is 0, you can ignore the weight penalty term. The idea here is that the goal of L1 regularization is to keep the magnitudes of the weight values small. If a weight is 0, its magnitude can't get any smaller. An example from the demo is:

```
if self.hoWeights[j,k] > 0.0:
  hoGrads[j,k] += lamda
elif self.hoWeights[j,k] < 0.0:="" hograds[j,k]="" -="">
```

Put a bit differently, during training, the back-propagation algorithm iteratively adds a weight-delta (which can be positive or negative) to each weight. The weight-delta is a fraction of the weight gradient. The weight gradient is the Calculus derivative of the error function plus or minus the L1 regularization constant.

Implementing L1 Regularization

The overall structure of the demo program, with a few edits to save space, is presented in **Listing 1**.

Listing 1: L1 Regularization Demo Program Structure

```
# nn_L1.py
# Python 3.x

import numpy as np
import random
import math

# helper functions
def showVector():
```

```
def showMatrixPartial(): ...
def makeData(): ...

class NeuralNetwork: ...

def main():
```

The hoGrads matrix holds hidden-to-output gradients. First, each base gradient is computed as the product of the associated output node signal and the associated input, which is a hidden node value. The computation of the output node signals isn't shown. Then, if the Boolean L1 flag parameter is set to True, an additional lambda parameter value (spelled as "lamda" to avoid a clash with the Python language keyword) is either added or subtracted, depending on the sign of the associated weight.

The input-to-hidden weight gradients are computed similarly:

```
for i in range(self.ni):
  for j in range(self.nh):
    ihGrads[i, j] = hSignals[j] * self.iNodes[i]
    if L1 == True:
      if self.ihWeights[i,j] > 0.0:
        ihGrads[i,j] += lamda
      elif self.ihWeights[i,j] < 0.0:="" ihgrads[i,j]="" -="lamda">
```

After using L1 regularization to compute modified gradients, the weights are updated exactly as they would be without L1. For example:

```
    # update input-to-hidden weights
    for i in range(self.ni):
      for j in range(self.nh):
        delta = learnRate * ihGrads[i,j]
        self.ihWeights[i,j] += delta
```

1 2 | next »

PRINTABLE FORMAT

Featured

Bridge to Kubernetes Simplifies Microservice Development in Visual Studio/VS Code

New Bridge to Kubernetes extensions available for Visual Studio and VS Code simplify microservice development by bridging a local dev machine to specific dependencies in remote clusters.

VS Code Favorite Python Closes In on Java in Popularity Ranking

The Visual Studio Code dev team was apparently on to something when it went all in on Python several years ago: It's poised to upend perennial No. 2 Java in the popular TIOBE Index of programming language popularity.

Binary Classification Using PyTorch: Preparing Data

Dr. James McCaffrey of Microsoft Research kicks off a series of four articles that present a complete end-to-end production-quality example of binary classification using a PyTorch neural network, including a full Python code sample and data files.

MOST POPULAR

.NET Interactive Now Does .NET Notebooks with VS Code

.NET Interactive users can now use a new Visual Studio Code Insiders experience to work with .NET Notebooks, in addition to other notebook options including Jupyter and nteract.

Code Small with C# in .NET nanoFramework for Embedded Systems

The .NET Foundation announced a new open source project under its direction, the .NET nanoFramework, which allows for C# coding in Visual Studio for constrained embedded devices.

MOST POPULAR

Free Webcasts

MOST POPULAR

Be A DBaaS Badass

Application Layer Attack Trends Through the Lens of Cloudflare Data

The Xamarin Summit

Xamarin 101: Why Xamarin for iOS and Android Development

> More Webcasts

Upcoming Events

Training Seminar: Virtual
September 30, 2020

Training Seminar: Virtual
October 15-16, 2020

Visual Studio Live! VirtCon
December 1-3, 2020

Training Seminar: San Francisco
December 10-11, 2020

San Francisco, CA

MOST POPULAR

Visual Studio Live! Las Vegas
February 28-March 5, 2021

Las Vegas, NV

Visual Studio Live! Austin
May 10-14, 2021

Austin, TX

Visual Studio Live! Microsoft HQ
August 9-13 2021

Most Popular Articles

MOST POPULAR

Death of the Dev Machine?

.NET Interactive Now Does .NET Notebooks with VS Code

Microsoft Says .NET 5 Replaces .NET Standard (Except for ...)

Blazor Guru Details Best Performance Practices

VS Code Favorite Python Closes In on Java in Popularity Ranking

Redmond, WA

Visual Studio Live! San Diego
September 12-16, 2021
San Diego, CA

Live! 360 Orlando
November 14-19, 2021
Orlando, FL

MOST POPULAR