Backpropagation Algorithm Report

Bryan Greener

2018-05-24

Implementation

This implementation of a neural network uses the backpropagation algorithm with gradient descent. The input data consists of numerous columns with both numerical and categorical data. To handle the categorical parameters, they were converted to one-hot arrays and re-inserted into the dataset. The input data was then normalized by dividing the dataset by the norm of the dataset which was obtained using Numpy's linalg.norm function. The denormalization process used later on simply reverses this operation. The requirements for this network included the use of two hidden layers. These hidden layers were set to 30 nodes and 10 nodes respectively. These node counts were determined using the following equation:

$$(i + j)^{0.5} + 10$$

The variables i and j represent the number of input layer nodes and the number of output layer nodes. Due to the low resulting number from the left side of this equation, 10 is added to the result. The input layer consists of a number of nodes equal to the amount of training parameters and the output layer consists of a single node resulting in a float value representing a normalized prediction of the price of a house. The dataset was split into training and testing subsets where the training set is 75% of the complete dataset and the testing set is 25%.

Algorithms used in this implementation include the sigmoid function, a sigmoid prime function, a feed forward algorithm, and the backpropagation algorithm. The sigmoid and sigmoid prime functions are as follows.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1}$$

$$\sigma'(z) = \frac{e^{-z}}{(1 + e^{-z})^2} \tag{2}$$

In the case of this program, z is a matrix corresponding to the input activations at any given layer in the network. Next, the feed forward algorithm takes in an $n \times 1$ matrix where n is the number of input parameters for the network. In this implementation the activation matrices a and z and the weight matrix w are stored in a class object but for the purpose of showing the algorithm they are included in the algorithm below. The other main algorithm involved is the backpropagation algorithm which is also listed below.

Algorithm 1: Feed Forward algorithm for sending matrix of inputs into network and getting single value result.

function FeedForward (a, z, w, x);
Input : Activation matrices a and z, weight matrix w, input matrix x.
Output: $\hat{y}$
$z[0] = x \cdot w[0]$
$a[0] = \sigma(z[0])$
for ($i = 1;\ i < l;\ i+ = 1$) {
    $z[i] = a[i-1] \cdot w[i]$
    $a[i] = \sigma(z[i])$
}
return $a[-1]$

Algorithm 2: Backward propagation algorithm for sending the errors calculated at the last layer back to the start of the network, updating weights along the way.

function Backprop (a, z, w, x, y, η);
Input: Activation matrices a and z, weight matrix w, input matrix x, output matrix y, hyperparameter η.
$\lambda = 1e-4$
$\delta = -(y - \hat{y})\sigma'(z[-1])$
for ($i = l-1;\ i > 0;\ i- = 1$) {
    $\Delta = \dfrac{a[i-1]^T \cdot \delta + \lambda w[i]}{len(x)}$
    $\delta = \delta \cdot w[i]^T \sigma'(z[i-1])$
    $w[i] = w[i] - \eta\Delta$
}
$\Delta = \dfrac{x^T \cdot \delta + \lambda w[0]}{len(x)}$
$w[0] = w[0] - \eta\Delta$

Algorithm 2 adds two new hyperparameters λ and η. η is the learning rate of the network and it scales the size of each step during the training process in order to allow more fine tuning of the behavior of training. λ is a very small value that is used to scale the weights to prevent overflow errors in the sigmoid functions.

Analysis

This implementation of a neural network using backprop did not include things such as momentum, smoothing, or mini-batches. Thus the accuracy of this network is not as high as it could be. After testing with different learning rates and network designs, this network was able to get an average error of $\pm\$40,000$ over hundreds of training epochs while using a 75/25 split of training and testing data. This accuracy improved when increasing the percentage of training data however this shows that the network was not able to generalize very well. Many times while testing this network on different randomizations of training data, the accuracy would be far worse than $\pm\$40,000$. This is due to the network getting stuck in local minima since no momentum was used. Along with this, the network's accuracy is lowered by the input data itself since many parameters of the dataset were ordinal and were handled simply as categorical. It's also likely that there are parameters that are hurting the network's overall accuracy or may just not be needed at all. By using statistical methods of reducing the noisy data, the network could improve both in time and accuracy however this was not performed in this implementation.

This dataset was also run through a different implementation designed for a class in spring 2018 to be used with the MNIST handwritten digit dataset. This version of the program uses RMSProp with Nesterov momentum. These two additions combined halved the prediction error and improved the running time of the network by a significant amount as well. Still though the overall error was $\pm\$20,000$ which has a lot of

room for improvement. Overall, the limiting factor to both networks is the input dataset and noisy data.