Backpropagation

Sean Wade

OVERVIEW

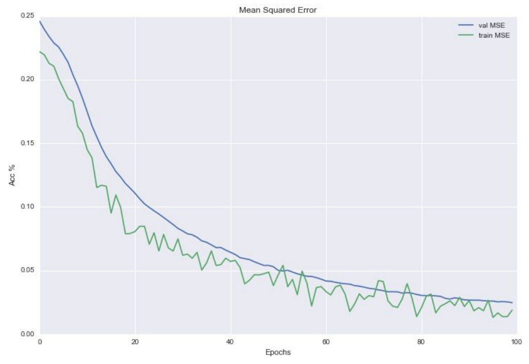
Backprop is the key element in training multi-layer perceptrons and the modern neural networks. This employs the idea of minimizing a loss function with gradient descent and calculating the gradient using backpropagation and automatic differentiation.

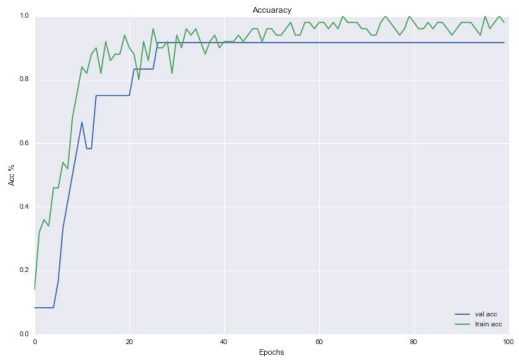
1. Implement Backprop

I chose to use python to implement backprop for a three layer neural net. For my implementation I created a modularized approach where a network is simply a stack of layer classes. Each fully connected layer can be given an arbitrary amount of nodes. The primary methods for these layers are the forward pass and backward pass through the data. The forward pass takes in an input and gives the output. The backward pass takes the upstream gradient and cached forward pass computations and then passes the gradient with respect to its weights down. The init method takes in various hyperparameters such as number of epochs, learning rate, batch size, and an early stopping criteria. The model with automatically terminate if it goes a determined number of epochs without improvement.

2. Iris Classification

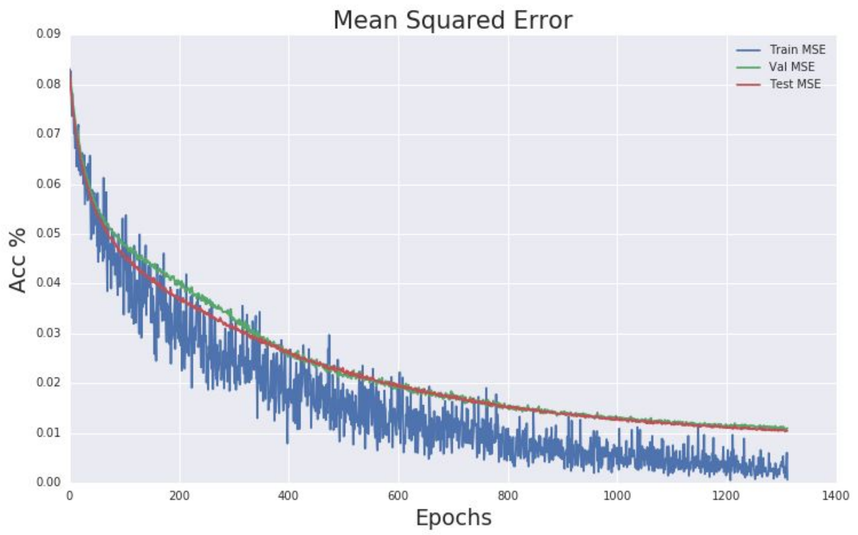
I first used my model for flower prediction on the Iris dataset. This dataset contains 150 instances of flowers. Each of which have four features and fit into one of three classes. To train it I used a ¾ split of the data, 8 hidden nodes, and learning rate .1. This is a very simple problem and my model was able to get 94% accuracy. These figures show the mean squared accuracy over time





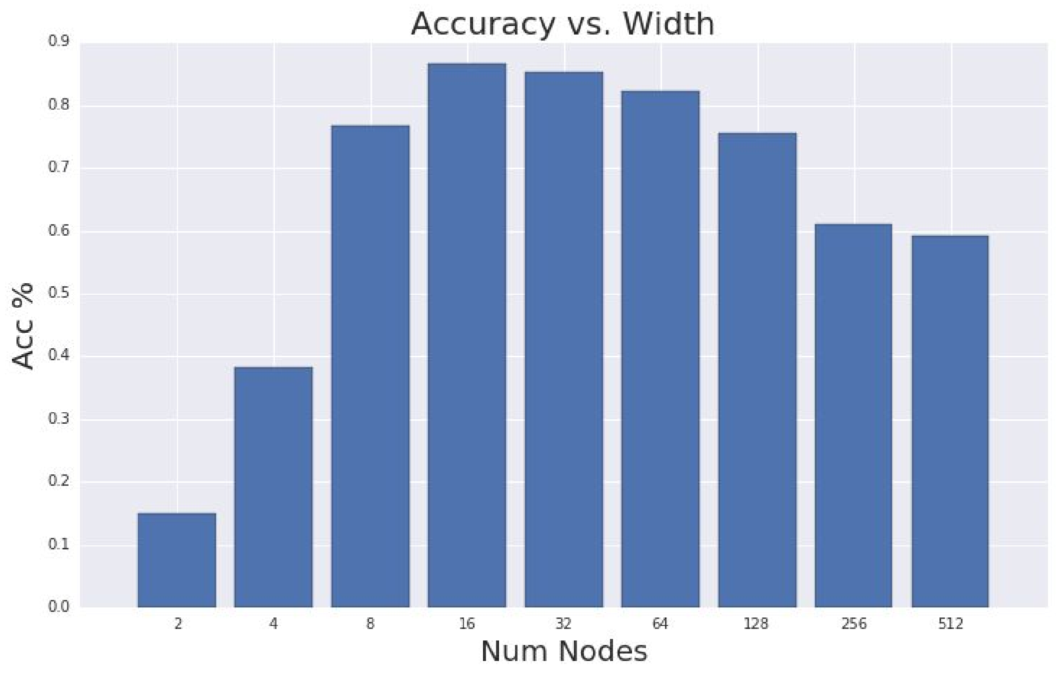
3. The Vowel Dataset

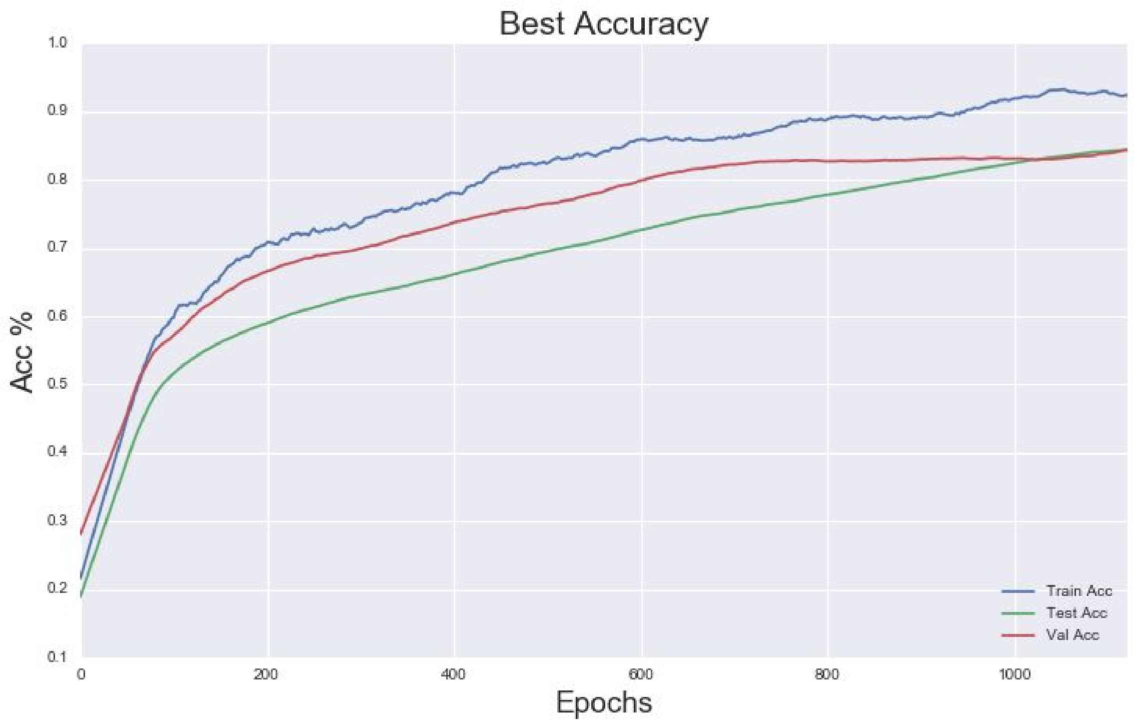
The next dataset I trained my neural network on was the vowel dataset. This dataset was more complex with 11 features and 10 prediction classes. For this it was important to normalize my inputs and use all the available features (especially since they are unlabeled). On my first run I was able to get 62% accuracy, which is much better than the baseline 9%. To improve my results I did cross validation on the learning rate of my stochastic gradient descent. I used the rates [.001, .01, .1, 1, 10] and averaged the results. This revealed that the best rate is .01. The graphs below show the mean squared error of the neural net and also the difference of learning rates.



4. Width of Hidden Layer

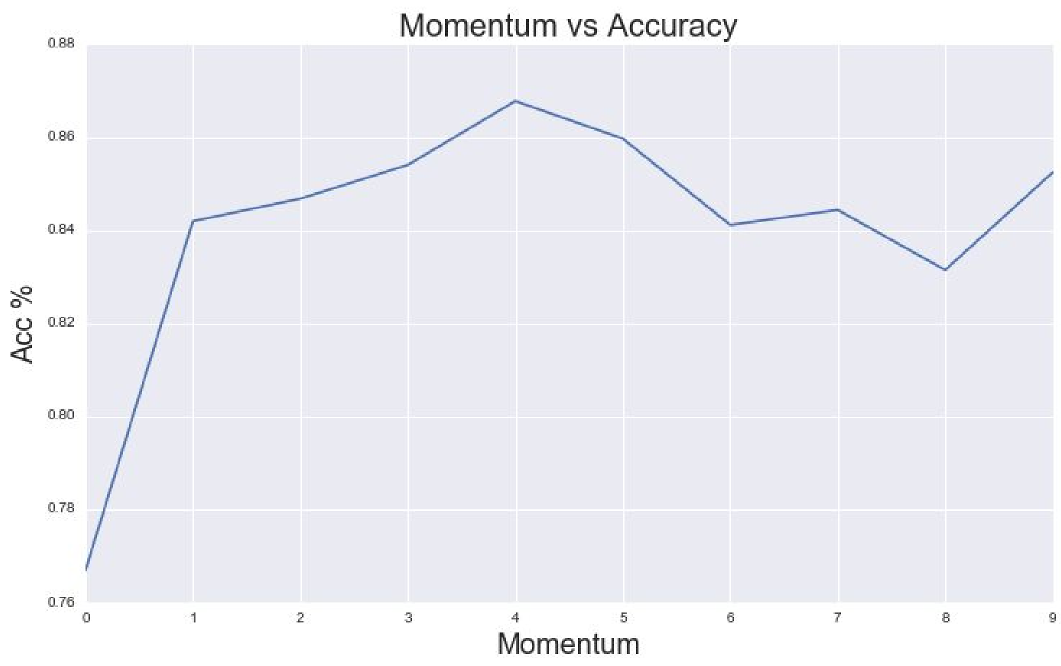
The next hyperparameter I optimized is the width of the hidden layer in the network. Each accuracy is the result for the average of that number of hidden layers. These experiments show 16 to be the optimal number. Any larger than that has diminishing returns and longer computation time. The graph directly below indicates this and the other shows the accuracy.





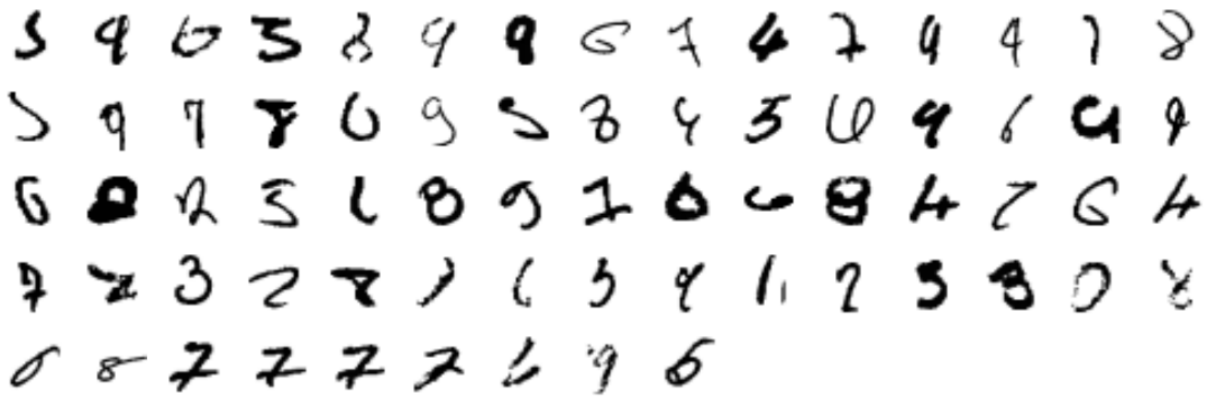
5. Voting Classification

The previous experiments used vanilla gradient descent to find the optimal weights. To improve this we will use a momentum term. This is analogous to momentum in the newtonian sense where continually moving in a direction builds up its momentum in that direction. This is useful for training faster through shallow ravines. The figure below shows that the best setting for momentum is .4.

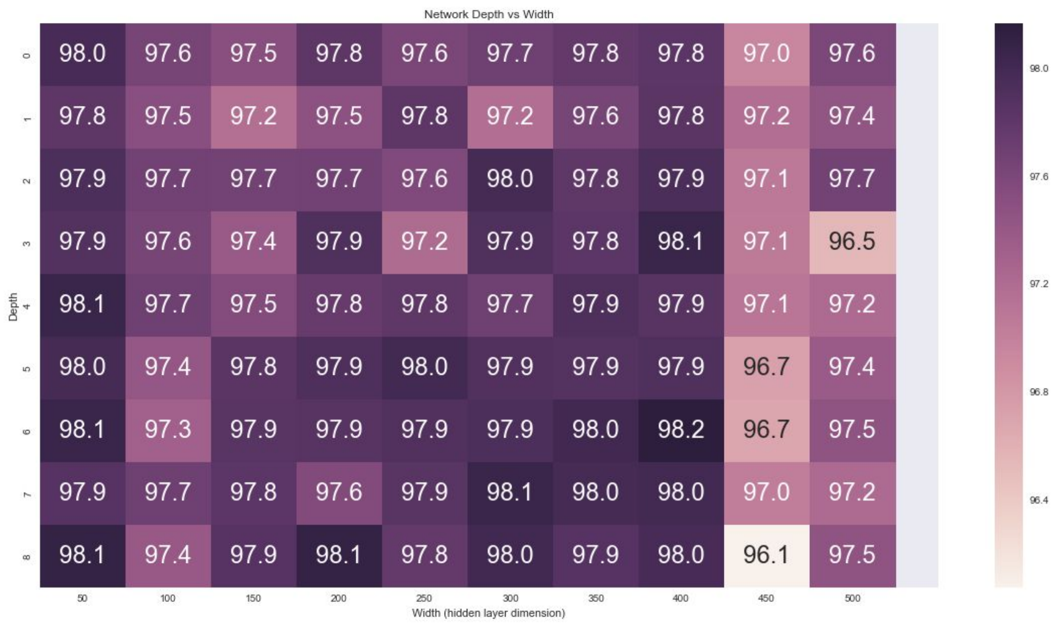


6. My Experiment

For my experiment I extended what we learned in backprop to different types of layers and functions. Specifically I implemented a convolutional neural network. This is very similar to the standard net above, however it has a slightly different topology and makes use of the convolution function. The problem I solved with this network was the MNIST handwritten dataset. This dataset consists of 60000 handwritten digits from zero to nine and we want to predict what the correct label is. The network has two convolutional layers, a fully connected layer, and a softmax. I was able to achieve 91% accuracy. Below are the difficult numbers that the algorithm was not able to classify. Looking at these it is not hard to see why it would have trouble.



After using my own tools I went further by doing the same thing using the deep learning framework Keras. I enjoy how simple and expressive their syntax is and was able to quickly develop a neural net that implements backprop. The goal of this image was to see the relation between depth and width of a network. These results were using many tricks such as dropout, batch normalization, and regularization. Since it was such a big job to run I did it on the BYU supercomputer. It is interesting to note that for a simple problem like MNIST that it is able to reach a peak with almost all layers. The bad results tend to occur after 400 width hidden layers .



Depth	50	100	150	200	250	300	350	400	450	500
0	98.0	97.6	97.5	97.8	97.6	97.7	97.8	97.8	97.0	97.6
1	97.8	97.5	97.2	97.5	97.8	97.2	97.6	97.8	97.2	97.4
2	97.9	97.7	97.7	97.7	97.6	98.0	97.8	97.9	97.1	97.7
3	97.9	97.6	97.4	97.9	97.2	97.9	97.8	98.1	97.1	96.5
4	98.1	97.7	97.5	97.8	97.8	97.7	97.9	97.9	97.1	97.2
5	98.0	97.4	97.8	97.9	98.0	97.9	97.9	97.9	96.7	97.4
6	98.1	97.3	97.9	97.9	97.9	97.9	98.0	98.2	96.7	97.5
7	97.9	97.7	97.8	97.6	97.9	98.1	98.0	98.0	97.0	97.2
8	98.1	97.4	97.9	98.1	97.8	98.0	97.9	98.0	96.1	97.5