Handwritten Digits Classification using Multi-layered Neural Networks

Sriram Ravindran
A53208651
sriram@ucsd.edu

Ojas Gupta
A53201624
ogupta@ucsd.edu

Abstract

Here in this project we have implemented a Multilayer Perceptron for 10-class classification. The MLP was trained and tested on famous handwritten MNIST dataset via gradient descent. We study the effect of Nesterov momentum on our model as well as consider other optimizations including a modifier tanh activation, mini-batch training. We also consider initializing the weights based on number of neurons in the layer. We also study the effect of number of hidden neurons and hidden layers on performance.

1 Classification using 2-layer Neural Network

We implement and study a simple 2-layer NN on Python. We used a helper function to load MNIST data. The data was normalized by diving with 256 and subtracting the mean pixel values. The output layer was a softmax layer but the hidden neurons used sigmoid activation function.

1.1 Checking the gradient

Cross-entropy error is used in our implementation. We added and subtracted a small perturbation ϵ to each of the weights in layers L1 and L2 one by one and divided the difference in resulting cost by epsilon. This is a first order approximation to the differential of the error, and is given by,

$$\frac{\partial E^n}{\partial w} = \frac{E(w+\epsilon) - E(w-\epsilon)}{2\epsilon}$$

We used $\epsilon = 0.01$ for the approximation and absolutes were averaged the results we obtained. It is expected that the error is within $O(\epsilon^2)$. Results we obtained are presented in Table 1. We see that the results are within expected range.

Table 1: Difference in gradient and first order approximation, Δ

Difference	Value
Mean Δ	7.5e-05
Max Δ	6e-04
Min Δ	5.4e-09

1.2 Implementation of 2-layer neural network

We implemented a 2-layer neural network with 100 sigmoid hidden neurons and a softmax layer. Backpropagation was used as the training algorithm for the model. The models were trained with 50000 training samples over 300 epochs and were validated using a validation set of 2000 examples. We didn't do early stopping, instead We collected testing results using weights obtained every time validation accuracy peaked higher than previous best.

Over 300 training epochs, the training, validation and testing accuracy kept increasing. In the end, we obtained 87.3% training accuracy and 84.2% testing accuracy with no additional tweaks. A plot of the training, validation and testing accuracies is presented below.

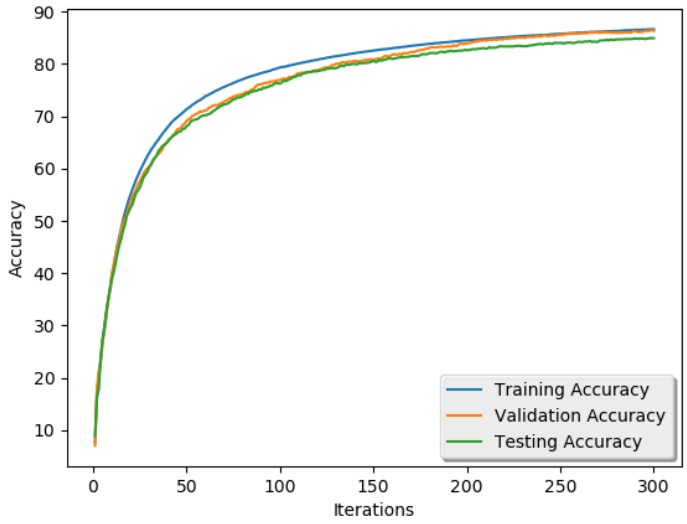

Figure 1: Plot of accuracy vs iterations.

2 Tricks of the Trade

Here in this section, as told in the problem statement we have read the sections 4.1-4.7 from the paper lecun98efficient.pdf. We have implemented the following ideas incrementally so as to get a clear cut understanding of the results we obtain.

2.1 4a Mini-batch training

We selected 1000 samples incrementally for each training epoch. We observed that the network took a lot more number of iterations to converge but converged faster in terms of time. Over several repeated tests, we found that eventually, after long number of iterations, the accuracy approaches full batch accuracy.

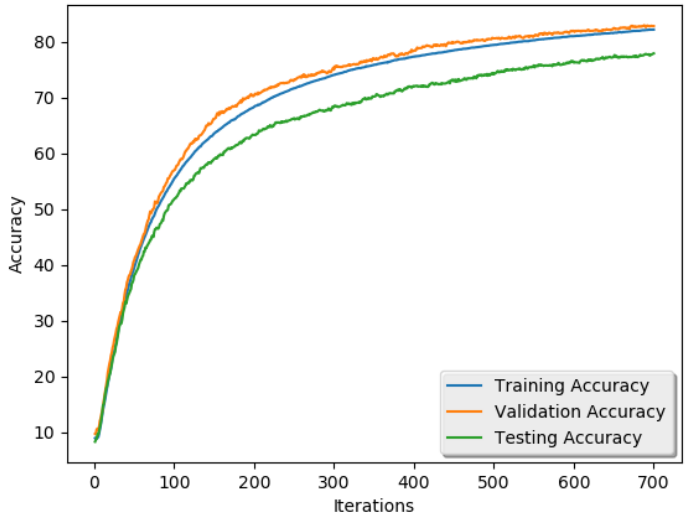

Figure 2: Plot of accuracy vs iterations in mini batch training.

2.2 4b Normalization

Followed normalization as performed in the previous question. Normalization helps obtain solution faster as weights are in similar range. We divided the images by 256 and subtracted the mean value from each of the pixels.

2.3 4c Funny Sigmoid

We implemented the variant of tanh that Prof. Gary calls Funny Sigmoid as presented below,

$$\sigma(x) = 1.7159 * tanh(2x/3)$$
$$\sigma'(x) = 1.7159 * (1 - 4x^2/9)$$

Following graph indicate the effect after using funny sigmoid.

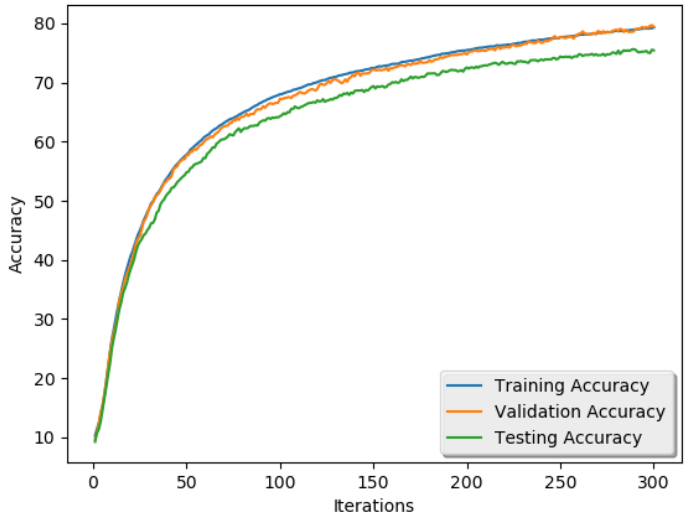

Figure 3: Plot of accuracy vs iterations.

They seem to perform similar to sigmoid over several trials. However, their power lies in the fact they provide zero mean and unit variance for the next. For this, we need to consider part 4d.

2.4 Question 4(d)

Successfully initialized the the weights with mean 0 and standard deviation 1/sqrt(fan-in), where fan-in is the number of inputs to the unit.

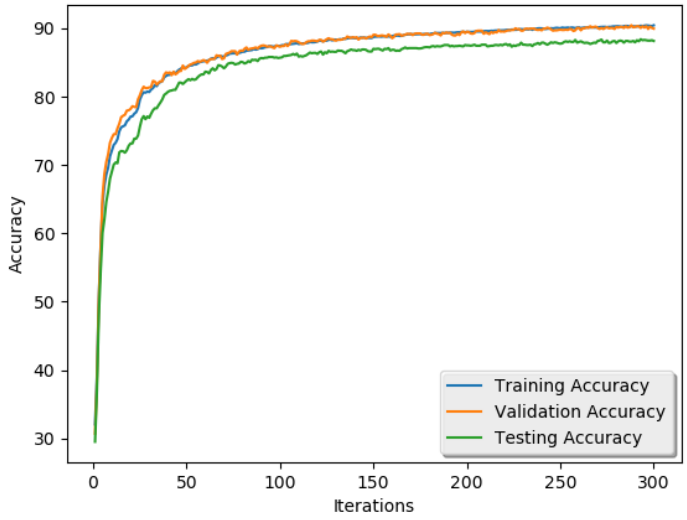

Figure 4: Plot of accuracy vs iterations - Funny sigmoid with weight initialization.

Above graph explain the difference we observe after initializing the weights in the prescribed manner.

2.5 Question 4(e)

Here we used Nesterov Momentum with momentum factor as 0.9 so as to update the weights of our network.

Following graph explains the effect of using this momentum feature.

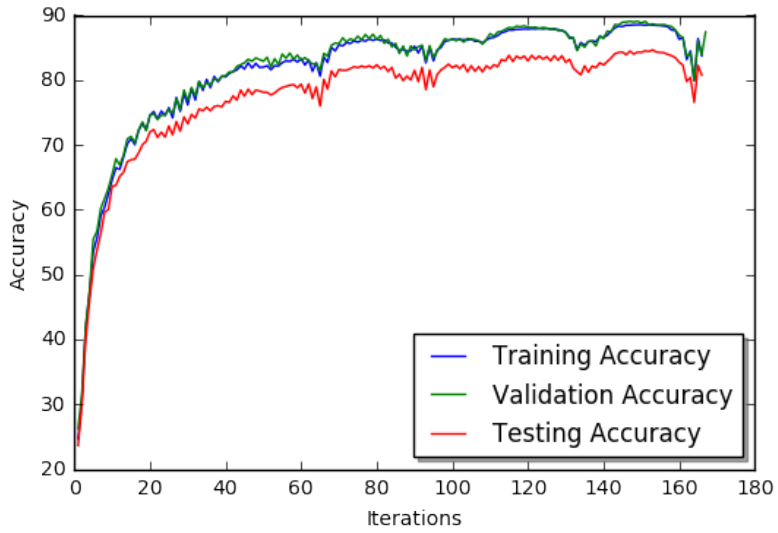

Figure 5: Plot of Training, Testing, Validation accuracy vs iterations using Nesterov momentum).

2.6 Question 4(f)

After using all the tricks of the trade, as I have shown in the graphs we conclude that the performance has been enhanced in terms of learning speed. Since we reached a level of accuracy in less number of iterations compared to the earlier model. It takes the sigmoid version 100 iterations to reach 80% accuracy and only about 50 iterations with optimization to achieve the same level.

3 Q5: Experimenting with Network Topology

Note: I have implemented both the parts of this problem after using the tricks of the trade from previous section. Therefore the code is in the configuration of Q4 i.e. hidden activation function is funny sigmoid, Nesterov Momentum is used with 0.9.

3.1 Question 5(a): Increasing Hidden Units

Here, as instructed in part(a) we have used 4 hidden units in our hidden layer and observed the following pattern as presented in the graph. We have doubled and halved the number of hidden units so as to check the pattern of the network training.

Here we observe that as the hidden units increase testing accuracy increases and becomes better.

If the number of hidden units is too small like 1-2 then the accuracy is also very small as shown in the graph. On the other hand if the hidden units are large then accuracy also increases as shown in the figure below.

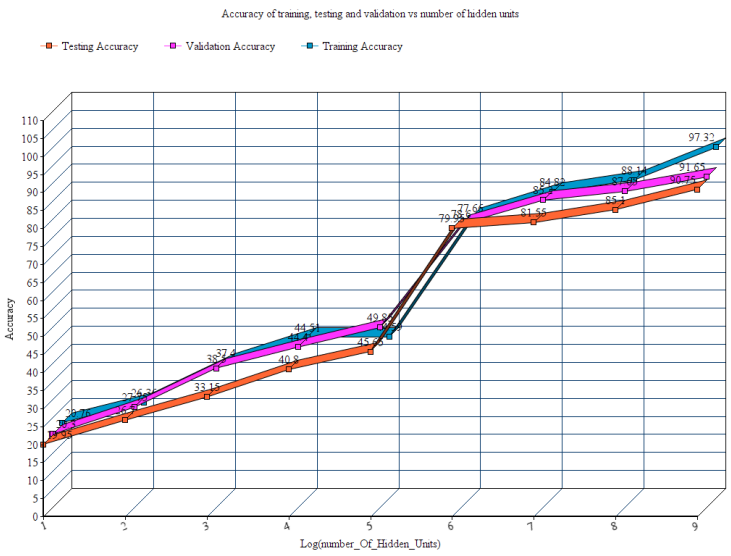

Figure 6: Plot of Training, Testing, Validation accuracy vs log(hidden-units).

3.2 Question 5(b): Increasing the hidden layer from 1 to 2

Here I have added another hidden layer with sigmoid "activation function. Here the hidden units in the first layer is 80 and the hidden units in the second hidden layer is 140. I have chosen the hidden units so as to equate the number of weights and biases as much as I can. Here learning rate is 0.00001 and Nesterov Momentum is used.

Here we see that the training speed has increased tremendously as previously when we used only one hidden layer we reached testing accuracy in 80s in 300 iterations

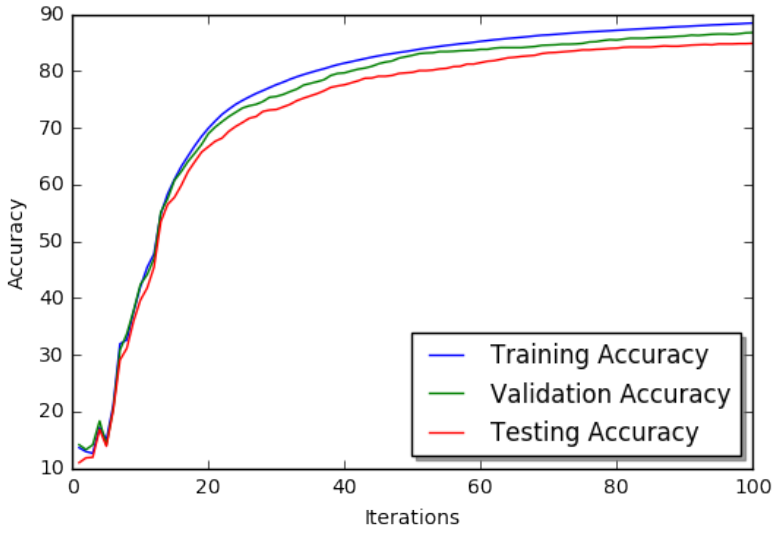

Figure 7: Plot of Training, Testing, Validation accuracy vs iterations.

4 Final observations

Multi-layered Neural Network is implemented in Python and used in classifying MNIST dataset. Here we used both the combination of one hidden layer and two hidden layer in our network model and found out that 2 hidden layer works better than 1 hidden layer in terms of accuracy and speed. Also we studied about different tricks of the trade and their affect on accuracy. We have also studied the importance of hidden units in training the weights. Hopefully our future work will involve in classifying images with this model.

5 Contribution

5.1 Ojas Gupta

Did some parts in question 4. Did Problem 5 and also contributed towards report writing. Also contributed towards bug fixing. Work was distributed equally.

5.2 Sriram Ravindran

I wrote the code for Problem 3. Did some parts in questions 4. Then work was taken up Ojas. Latex work was also distributed almost equally. We did a lot of pair programming.