CSE 253 HW2

Chao Long
Department of Electrical and Computer Engineering
University of California San Diego
c3long@eng.ucsd.edu

Nemil Shah
Department of Computer Science
University of California San Diego
nbshah@eng.ucsd.edu

Abstract

In this paper we discuss a three layer Neural Network model to detect handwritten digits from images. We have trained our model on the MNIST dataset. We implemented backpropogation to make our model learn and use gradient descent to update our weights. We got an accuracy of **97.35%** by implementing a simple backpropogation model which uses Sigmoid and Softmax activation units at the hidden and output units respectively. Furthermore, we improvised our model by using different tricks such as including momentum in the gradient, using tanh activation function at the hidden layer and a four layer network (2 hidden layers with sigmoid activation). Though the increase in accuracy was not much by adding the improvisations but the learning rate of the model was much faster and hence took lesser number of iterations.

1 Introduction

Handwritten number detection is the problem of training a learning model to be able to classify an image of a written number into one of ten classes, i.e. [0-9]. We have obtained the MNIST dataset of handwritten numbers from `http://yann.lecun.com/exdb/mnist/`.

A three layer neural network model was trained using 50,000 samples from the training set and then tested on 10,000 samples. The input layer consists of 784 neurons (one for each pixel in the 28x28 image). The hidden layer consists of 64 neurons and the output layer of 10 neurons (one for each class [0-9]).

The model performs two passes. The forward propogation where we use our learned weights to predict the output and then a backward pass where we porpogate the error from the output and our prediction back to the input layer. This method is called Backpropogation. Backpropagation is where we calculate the error contribution of each neuron after a batch of data is processed. We used Backpropogation to propagate the error from the output to the input layer and then update the weights accordingly.

In the process of training the models, given a set of parameters, we used gradient descent method to get the values of weights that would give us the least error on holdout set. We do not use the test set for measuring accuracy because in most real world cases we do not have access to the test data. But that being said, we show an analysis of how well this model has performed on the test set after training and see if the holdout set is a good representation of the test set.

2 Classification

In this section we describe how we have used the Backpropogation model in handwritten number detection by training the model on 50,000 samples from the MNIST dataset.

Submitted to 31st Conference on Neural Information Processing Systems (NIPS 2017). Do not distribute.

2.1 Method

First we loaded the MNIST dataset using the loader function `https://gist.github.com/akesling/5358964`, in Python. Then the following steps were followed to make the training, validation and test sets.

1. We extract the features from the 28x28 image from the dataset and make it a 784x1 feature vector.
2. The feature vector contains the brightness of each pixel of the image and hence the values range from [0, 255]. We normalize this range by dividing each value by 127.5 and then subtract 1 from them so now the values range from [-1, 1].
3. Then we shuffle the examples in the training set.
4. We divide the total training set into two sets, i.e. training and validation sets with 50,000 and 10,000 samples respectively.

Now we have a training, validation and test sets of 50,000, 10,000 and 10,000 samples respectively.

We then build a model consisting of three layers. The input, hidden and out layers with 784, 64 and 10 neurons, respectively, in each layer. We used the 'Sigmoid' activation function at the hidden layer and the 'Softmax' activation function at the output layer.

We use gradient descent on our cost function to learn our weights. The cost function to minimize is defined as,

$$E(w) = -\frac{1}{N}\sum_{n}\sum_{k=1}^{c} t_k^n ln y_k^n$$

Here, E(w) is the cross entropy loss function (cost), t_k^n is the target label of the image for the class k and y_k^n is the probability of it being in class c.

To perform backpropogation we calculated the gradient of the weights with respect to the loss function at every layer. We checked the gradients computed from backpropogation to that computed using the numerical approximation. The forumala to compute the gradient of a particular weight using numerical approximation is given as,

$$\frac{\partial E}{\partial w_{ij}} \approx \frac{E^n(w_{ij}+\varepsilon) - E^n(w_{ij}-\varepsilon)}{2\varepsilon}$$

where ε is a small constant. We have chosen $\varepsilon = 0.01$. We used a small dataset 50 samples from the training set. We then computed the difference for some weights and biases from the input to the hidden and hidden output layer. We observed that the difference was in 10^{-4}. We have tabulated our observations in the results section.

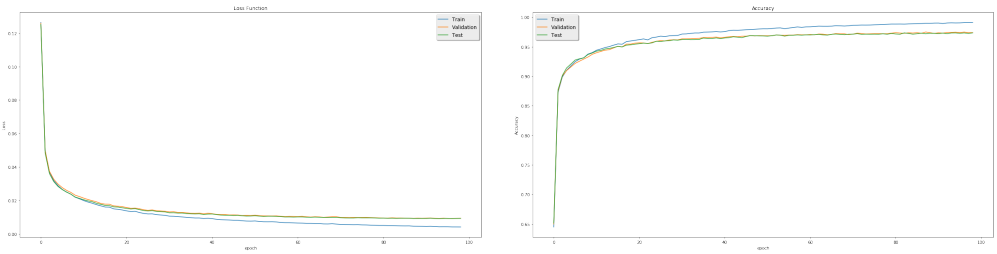
After checking, we updated the weights over the training. The output labels were converted to one-hot encoding. We used random initializations for the weights and biases and mini-batches of size 128 samples to update the weights. We used early stopping so that if our loss on the weights function increased for 3 continuous epochs then we would stop training. We also used annealing to reduce the learning rate over epochs.

2.2 Result

We used the training data to train our model and learn the weights and used the validation set to tune the hyperparamters like the no. of epochs, learning rate and the value of T used for annealing. The final values for $\eta = 0.1$ and for T=10000

We achieved an accuracy of **97.34%** on the test set. The final accuracy on the validation set was 97.49%.

The plots below (Figure 1) illustrate the loss and accuracies over the training, validation, and testing sets.



(a) *Loss* (b) *Accuracy*

Figure 1: Loss and Accuracy plots v/s no of iterations

The Table 1 below also represents the difference in the weights and biases calculated using Backpropogation and numerical approximation over the two layers. In the table Weight[x][y] represents the weight connected from the 'x' neuron in one layer to the 'y' neuron in the other layer. Similarly the bias[x] represents the bias added to the weights associated with the 'x' neuron in the other layer.

	Input to Hidden	Hidden to Output
Weight[1][1]	0.0001556	0.0001515
Weight[5][5]	0.0007109	0.0006898
Weight[10][10]	0.0005960	0.0004965
Weight[15][10]	0.0003006	0.0004613
Weight[20][5]	0.0001521	0.0001980
bias[1]	0.0004342	0.0008660
bias[5]	0.0007344	0.0001741
bias[10]	0.0006406	0.0004545

Table 1: Difference in derivative of weights

3 Adding Tricks of trade

3.1 Method

(a) We already implemented shuffling of the mini batches after each epoch in our previous model with which we achieved an accuracy of 97.34%.

(b) We used the 'tanh(x)' activation function at the hidden layer instead of the sigmoid used in the previous model. The activation used was '1.7159tanh(2x/3)'. We observe that the accuracy on the testing set does not improve much but the no of iterations it takes to achieve a similar accuracy has reduced by a lot. The accuracy achieved was 97.24% but it only took around 50 iterations which is less as compared to the no of iterations (around 100) for the sigmoid activation function.

The derivative of the tanh activation function is given as,

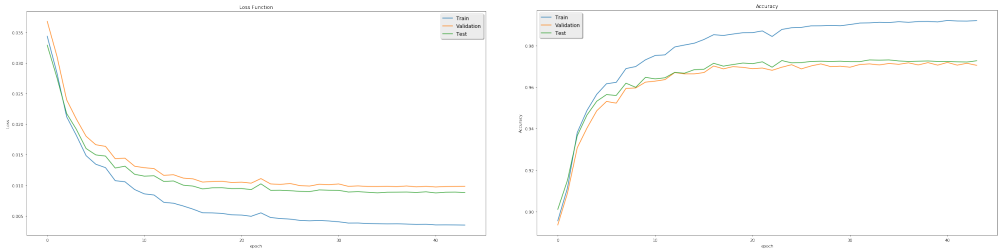
$$\frac{\partial}{\partial x}[1.7159 * tanh(\frac{2x}{3})] \quad = 1.7159 * \frac{2}{3} * [1 - tanh^2(\frac{2x}{3})]$$

(c) After using the tanh function and initializing the weights from the normal distribution, of zero mean and standard deviation of 1/sqrt(fan-in), where fan-in is the number of input to the unit, we get an improved accuracy of 97.31%. Also the no if iterations it takes to achieve this is reduced to 40 iterations. We initialize the weights from a distribution of 0 mean and unit standard deviation so that weighted sum of the input is in the linear range and the gradients will be the largest and hence the model can learn the linear part of the model first and then the non-linear part.

(d) We implemented momentum in our gradient descent with an $\alpha = 0.9$. We were able to achieve an improved accuracy of 97.38% within just 25 iterations. We have plotted our loss and accuracies against the 25 epochs in the results section.

3.2 Result

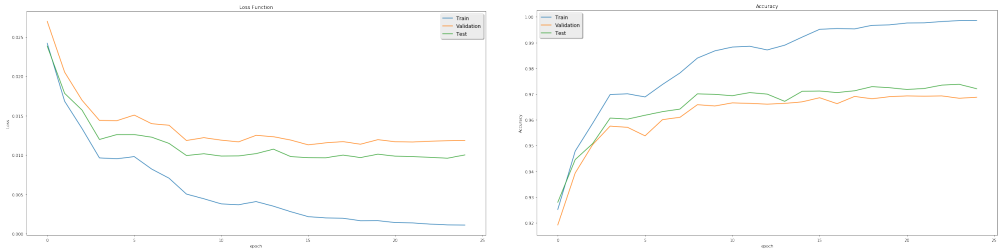
The plots below (Figure 2) illustrate the loss and accuracies over the training, validation, and testing sets after using a tanh activation function and normal initialization of weights instead of the sigmoid.



(a) *Loss* (b) *Accuracy*

Figure 2: Loss and Accuracy plots v/s epochs (tanh activation and normal initialization of weights)

The plots below (Figure 3) illustrate the loss and accuracies over the training, validation, and testing sets after using momentum for gradient descent. As seen the number of iterations have reduced to only 25.



(a) *Loss* (b) *Accuracy*

Figure 3: Loss and Accuracy plots v/s no of iterations (gradient descent with momentum)

4 Experiment with Network Topology

4.1 Method

In this part, we try to experiment with network topology. Firstly, the number of hidden layers are changed to 32 and 128 separately. Then we try to increase the number of hidden layers. In order to get the same number of parameters as our previous network, the new number of hidden units to double hidden layer is set to 60 after calculation. The calculation method is below,

Assuming the number of units in double hidden layers is x, in order to get the same number of parameters, we can get,

$$784 * 64 + 64 * 10 + 64 + 10 = 784 * x + x * x + x * 10 + x + x + 10$$

$$x \approx 60$$

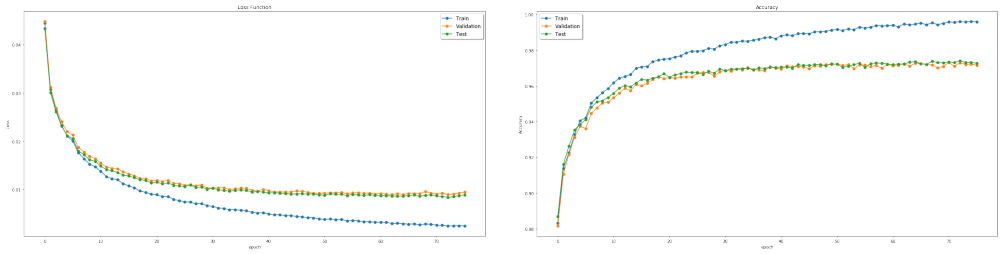
Finally, we also tried several new topologies like adding more hidden units, Nesterov momentum and Relu units to increase the testing accuracy.

The initial learning rate η here is 0.05 and the input weights are initialized to 0 zero mean and standard deviation 1/sqrt(fan-in). Besides, the activation function is simple tanh function below

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

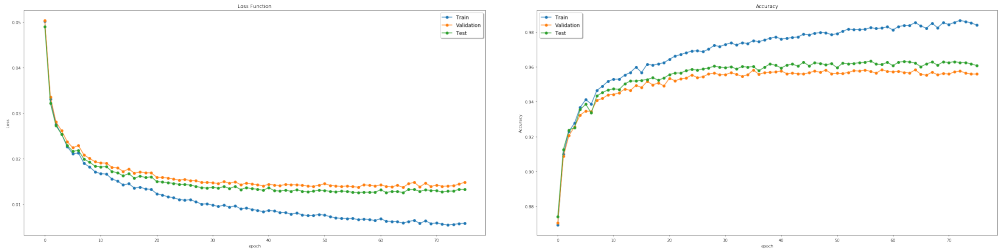
, and we also used simple momentum to get fast convergence before all the adjustments we have stated above.

4.2 Result



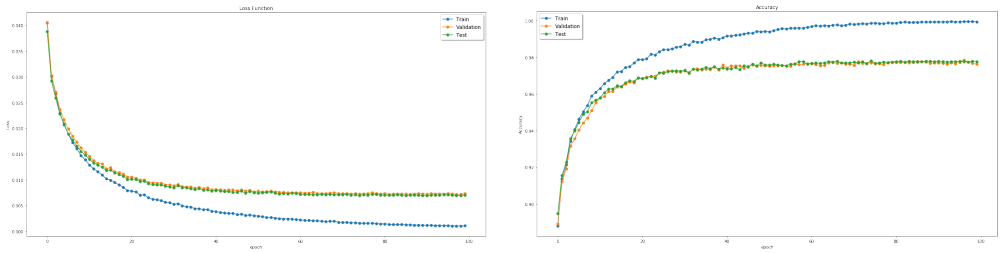
(a) *Loss for 64 hidden units* (b) *Accuracy for 64 hidden units*

Figure 4: 64 hidden units loss and accuracy



(a) *Loss for 32 hidden units* (b) *Accuracy for 32 hidden units*

Figure 5: 32 hidden units loss and accuracy

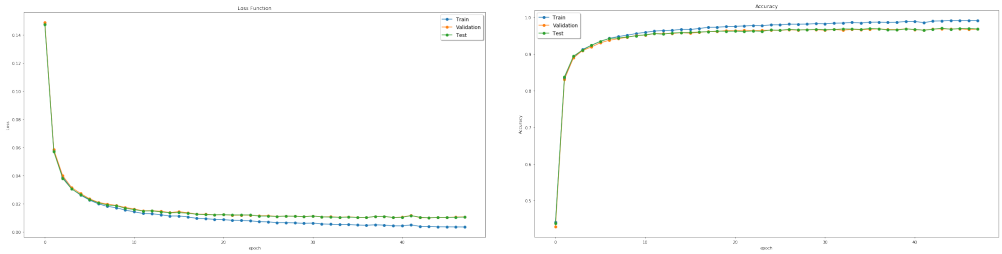


(a) *Loss for 128 hidden units* (b) *Accuracy for 128 hidden units*

Figure 6: 128 hidden units loss and accuracy

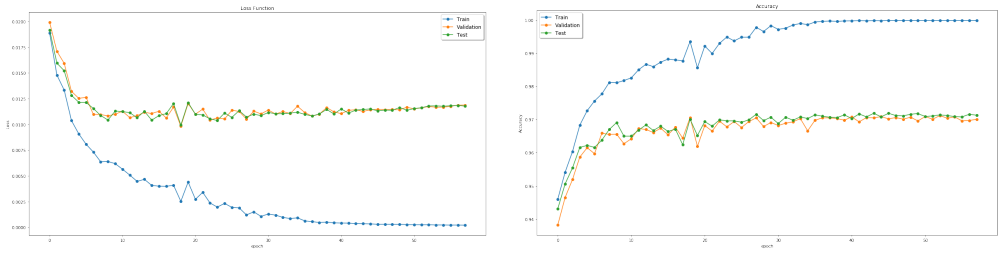
4.3 Discussion

As we can see in Figure 5 and Figure 6, if we just half the number of hidden units to 32, the training stops earlier and the accuracy(**96.25%**) is worse than 64 hidden units(**97.41%**). In my point of view, as we half the hidden units, the architecture of the neuron network seems to be simple and it easily suffers from overfitting problems. However, when the number of hidden units are doubled to 128, the



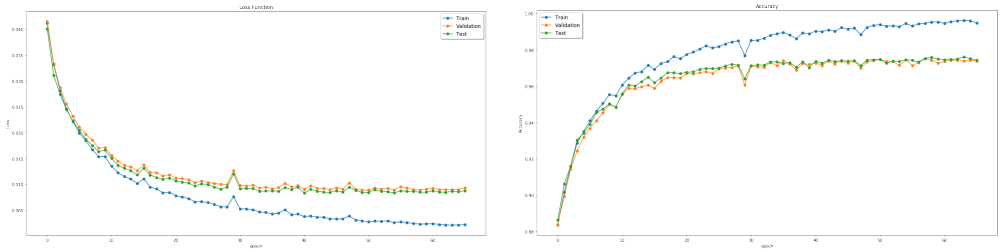
(a) *Loss for double 60 hidden units* (b) *Accuracy for double 60 hidden units*

Figure 7: Double 60 hidden units loss and accuracy



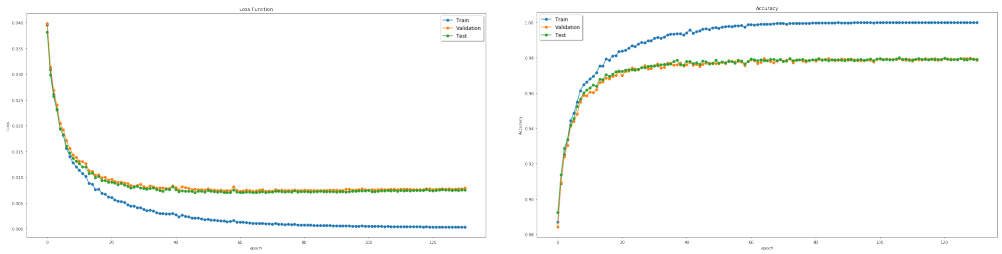
(a) *Loss for Nesterov monmentum* (b) *Accuracy forNesterov monmentum*

Figure 8: Nesterov monmentum 64 hidden units loss and accuracy



(a) *Loss for relu* (b) *Accuracy for relu*

Figure 9: Relu 64 hidden units loss and accuracy



(a) *Loss for relu* (b) *Accuracy for relu*

Figure 10: Relu 200 hidden units loss and accuracy

accuracy of the testing set(**97.77%**) becomes higher than our previous result in 64 hidden units. This is reasonable since the more hidden units we have, the more features we will need to learn, thus our model seems to be more optimal to get better accuracy result for the testing set when we have enough data.

From Figure Figure 7 , we can see the accuracy(**96.76%**) is almost the same as 64 hidden units(**97.41%**) . And we can also observe early stopping problems like Figure 4 for 64 hidden units. This is reasonable since we have the same number of parameters, and our models seems to learn the same complicated features, thus the accuracy and loss results seem to be the same even if we add more hidden layers.

In Figure 8, we use Nesterov monmentum method to train our model with 64 hidden units. As we can see, the accuracy is **97.09%**, not higher than before, and it also gets the overfitting issue. Besides, the loss and accuracy seem to get more fluctuation while using Nesterov monmentum and the training becomes much faster like we have used monmentum before.

In Figure 9 , we use relu activation function to train the dataset. The accuracy is **97.5%**, a little bit higher than tanh function for 64 hidden units we have used before.

In Figure 10 , we increase the number of hidden units to 200 and use the relu activation function simultaneously. The accuracy is **97.88%**, a little bit higher than 128 hidden units we have used before. This is optimal, since more features can be trained well, leading to a better testing accuracy by our large enough dataset.

5 Conclusion

We have successfully implemented a model which uses Backpropogation to learn the features in a given image to predict the digit represented in that image. We improvised the algorithm by including different activation functions, weight initializations and modifications to gradient descent by including momentum. We experimented with the existing topology by changing the number of neurons and by changing the number of hidden layers. Our final model consisting one hidden layer with 200 neurons and 'ReLU' as it's activation function was able to achieve an accuracy of **97.88%** which is much better than a two layer network (around 92%).

6 Citations

`http://yann.lecun.com/exdb/mnist/`
`https://gist.github.com/akesling/5358964`

7 Learning outcomes

We primarily learned how can we add a hidden layer and propagate the error backwards using gradients. We implemented a two layer network in the last assignment and here we inserted one more layer and worked out the math to propagate the error back from the output layer to improve our weights. We learned how adding another layer helps the model in learning the internal features of the training set which in turn helps the model to make better predictions. We also learned different tricks to help the model converge faster. By experimenting with the topology we learned how reducing the number of neurons in the hidden layer the model is not able to learn enough features and thus does not provide a great accuracy. On the other hand by increasing the number of hidden layers we observe that the model can learn even more complex features which might not be that helpful here because the digits do have such intricate features but can surely help in a dataset with colored images and more features.

8 Collaboration

Most of the backpropgation model coding and reporting was done in pairs. Some of the internal tasks were then divided. Nemil took up the task to tune the parameters, implement gradient checking and some tricks of the trade. Chao experimented with the network topology by changing no. of neurons and adding layers and also, implemented the ReLU activation and Nesterov momentum.