CSE 253: Neural Networks

Programming Assignment 2

Multi-layer Neural Networks

Saksham Sharma (PID: A53220021)
Department of Computer Science and Engineering
University of California, San Diego
San Diego, CA 92093
sas111@eng.ucsd.edu

Kriti Aggarwal (PID: A53214465)
Department of Computer Science and Engineering
University of California, San Diego
San Diego, CA 92093
kriti@eng.ucsd.edu

Abstract

We explore the use of multi-layer neural networks for the task of handwritten digit recognition. MNIST is a dataset of handwritten digits, originally comprising of 60000 training examples and 10000 test examples. However, for the experiment we conducted, we sampled 50,000 training examples and used the rest 10,000 examples as the validation set. Working with the feedforward propagation and backpropagation algorithm, we built a classifier that maps each input data to one of the labels t∈{0,1,2,3,4,5,6,7,8,9}. We used both the batch gradient and stochastic gradient descent techniques for optimizing on the Cost Function along with early stopping. We also applied various "Tricks of the trade" techniques (discussed in detail in section 4) in order to speed up the optimization. Without using 'tricks of the trade' we achieved an accuracy of **96.46%** on the test set, and **97.868%** on the training set. Subsequently, after applying 'tricks of the trade' techniques, we achieved an accuracy of **96.7%** on test set and **99.608%** on training set.

3 Classification

3.1 Introduction

The goal in this problem is to classify the handwritten images in the MNIST database to the corresponding digits from 0 to 9, i.e. we have to map each input data to one of the labels t∈{0,1,2,3,4,5,6,7,8,9}. In order to achieve this task we have used the concepts of "Multi-layer Neural Networks". The architecture of a typical multi-layer neural network is shown in the figure below:

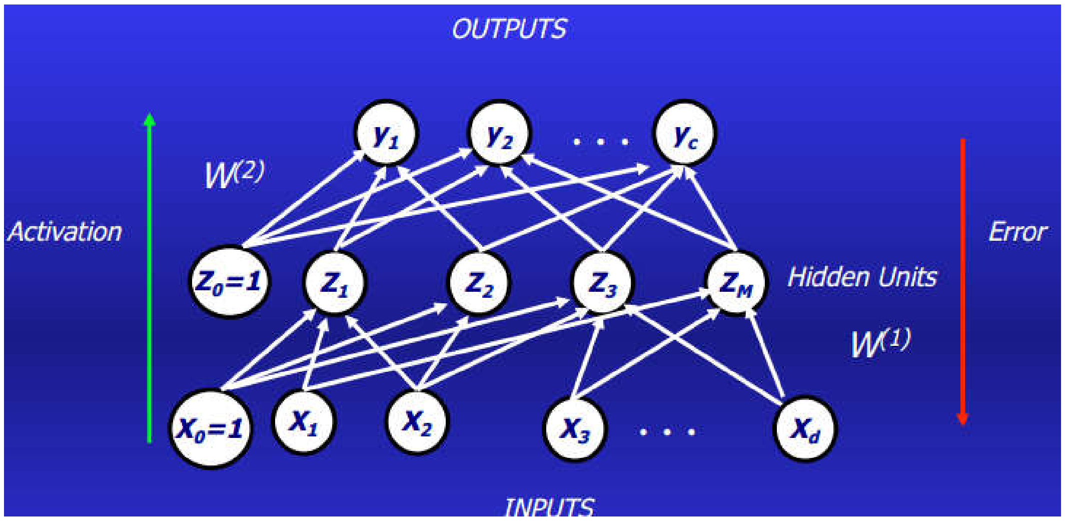

Fig 3.1 Architecture of a typical multi-layer neural network. (**This figure has been taken from the lecture slides.*)

There are two important concepts related to the multi-layer neural networks: (a) Feedforward propagation and (b) Backpropagation. They are described briefly in the following subsections.

3.1.1 Feedforward propagation

The feedforward propagation basically means that the information (activation) moves in only one direction, forward, from the input nodes, through the hidden nodes and to the output nodes. We have used the logistic activation function for the hidden layer. i.e.

$$y_j = \frac{1}{1 + e^{-a_j}}$$

where y_j is the output of the jth hidden unit in the hidden layer and a_j is the weighted sum of the inputs to unit j.

For the output layer, we have used the softmax activation function, i.e.

$$y(\boldsymbol{Z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

where $y(\boldsymbol{Z})_j$ is the output of the jth node in the output layer and z_j is the weighted sum of the inputs to the node j in the output layer.

3.1.2 Backpropagation

The backpropagation (backward propagation of errors) is the most common method of training artificial neural networks. In this algorithm, first the input vector is presented to the network and then it is propagated forward through the network, layer by layer (using the activation functions given in section 3.1.1), until it reaches the output layer. Then the output of the network is compared to the desired output using cross-entropy cost function which is defined as follows:

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

Here, t_k^n is the kth component of target vector t^n for example n and N is the total number of training examples. The factor $\frac{1}{N}$ normalizes the error over different training set sizes.

After that an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards, starting from the output, until each neuron has an associated error value which roughly represents its contribution to the original output [1]. Backpropagation then uses these error values to calculate the gradient of the loss function with respect to the weights in the network. Finally, this gradient is fed to the "Gradient Descent" learning algorithm which in turn update the weights so as to minimize the cross entropy cost function.

The weight update equations have already been derived in the Part 1 of this programming assignment. Please refer to the answers for part 1 for weight update equations.

3.2 Methods used

We used the following methods in order to complete this task:

1. **Normalization:** We divided the pixel values by 255 so that they are in the range [0,1]. Then we subtracted the mean over all of the pixels in each image, so that the input has zero mean.
2. **Batch gradient:** We used full batch gradient descent as the learning algorithm.
3. **Learning rate:** The initial learning rate used was 0.0001.
4. **Early stopping:** We used early stopping as the criteria to decide when to stop training. When the error on the validation set began to go up consistently over 4 epochs, we used the weights with the minimum error on the validation set as the final weights.
5. **Output unit activation:** We used softmax function as the output unit activation function.
6. **Hidden unit activation:** We used logistic function as the hidden unit activation function.

Classification of handwritten images from the MNIST database into digits (0 to 9) was performed using feedforward and backpropagation algorithms.

3.3 Results and discussion

(a) Read in the data from the MNIST database. You can use a loader function to read in the data. Loader functions for matlab can be found in http://ufldl.stanford.edu/wiki/resources/mnistHelper.zip. Python helper functions can be found in https://gist.github.com/akesling/5358964.

Answer) We used Python helper functions to read in the data from the MNIST database.

(b) The pixel values in the digit images are in the range [0..255]. Divide by 255 so that they are in the range [0..1], and then subtract the mean over all of the pixels in each image. I.e., if the mean pixel value in an image of a "4" is 0.3, you would subtract 0.3 from each pixel value.

Answer) We divided the pixel values by 255 so that they are in the range [0,1], and then subtracted the mean over all of the pixels in each image.

(c) While you should use the softmax activation function at the output level, for the hidden layer, you can use the logistic, i.e., $y_j = 1/(1 + exp(-a_j))$. This has the nice feature that $dy/da = y(1 - y)$.

Answer) We used the softmax activation function at the output level. For the hidden layer, we used the logistic function.

(d) Check your code for computing the gradient using a small subset of data - so small, you

can use one input-output pattern! You can compute the slope with respect to one weight using the numerical approximation:

$$\frac{d}{dw}E^n(w) \approx \frac{E^n(w+\epsilon) - E^n(w-\epsilon)}{2\epsilon}$$

where ϵ is a small constant, e.g., 10^{-2}. Compare the gradient computed using numerical approximation with the one computed as in backpropagation. The difference of the gradients should be within big-O of ϵ^2, so if you used 10^{-2}, your gradients should agree within 10^{-4}. (See section 4.8.4 in Bishop for more details). Note that *w* here is *one* weight in the network, so this must be repeated for *every* weight and bias. Report your results.

Answer) We used one input-output pattern for checking the code. The value of ϵ was equal to **10^{-2}**. The comparisons are as follows:

- Comparison for $\mathbf{w_{ij}}$ (i.e. weight from node i in the input layer to node j in the hidden layer)
  a) **Maximum**(|Gradient computed using numerical approximation - Gradient computed using Backpropagation|) = **3.12495556987e-07**
  b) **Mean**(|Gradient computed using numerical approximation - Gradient computed using Backpropagation|) = **1.47986095818e-08**

- Comparison for $\mathbf{w_{jk}}$ (i.e. weight from node j in the hidden layer to node k in the output layer)
  a) **Maximum**(|Gradient computed using numerical approximation - Gradient computed using Backpropagation|) = **5.2202338173e-07**
  b) **Mean**(|Gradient computed using numerical approximation - Gradient computed using Backpropagation|) = **1.59889126371e-07**

Thus, we observed that the difference of the gradients is within big-O of ϵ^2, i.e. 10^{-4}.

(e) Using the update rule you obtained from 2(c), perform gradient descent to learn a classifier that maps each input data to one of the labels *t*∈ *2 {*0, ..., 9*}* (but using a one-hot encoding). Use a hold-out set to decide when to stop training. (*Hint*: You may choose to split the training set of 60000 images to two subsets: one training set with 50000 training images and one validation set with 10000 images.) Stop training when the error on the validation set goes up, or better yet, save the weights as you go, keeping the ones from when the validation set error was at a minimum. Report your training procedure and plot your training and testing accuracy vs. number of training iterations of gradient descent. Again, by *accuracy*, we mean the percent correct on the training and testing patterns.

Answer)

- Using the update rule which we obtained from 2(c), we performed gradient descent to learn a classifier that maps each input data to one of the labels t ∈ {0,…,9}(using one hot encoding).
- We also used a hold-out set to decide when to stop training.
- The theory regarding the training procedure has been briefly discussed in section 3.1.
- The methods used for the training procedure have been discussed briefly in section 3.2.

The results are as follows:

Epochs: 1000

Accuracy Train: 97.868%

Accuracy Test: 96.46%

Accuracy Hold: 96.84%

Cross entropy error Test: 0.251247016989

Cross entropy error Hold: 0.289871554239

The plot for training and testing accuracy vs. number of training iterations of gradient descent is given below:

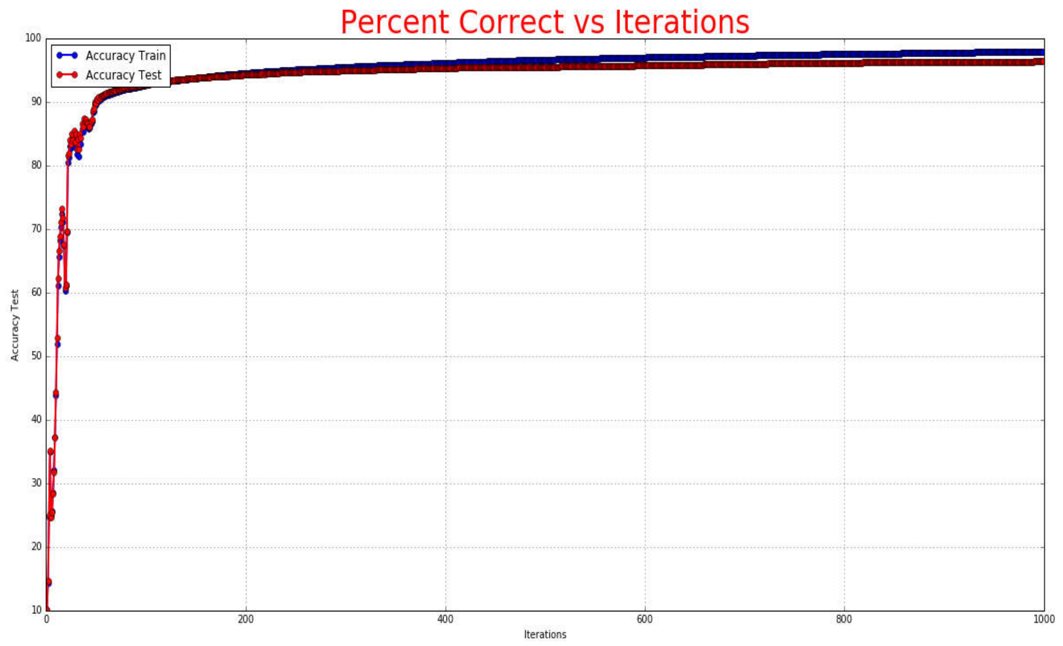

Fig. 3.2 The plot for training(blue) and testing(red) accuracy vs. number of iterations.

4 Adding the "Tricks of the Trade."

4.1 Introduction

There are a number of tricks that can enhance the learning rate and performance of neural networks. In this section, we will be discussing some of those tricks.

1. **Shuffling the examples**
   Since, the network learns fastest if it is presented with the most unexpected data. Therefore, in stochastic gradient descent, it makes more sense to shuffle the examples before performing the gradient descent. For the best results, in mini batch version of the stochastic gradient descent, each mini batch should have balanced samples from all of the classes in the output.

2. **Stochastic mini batch gradient descent**

   In case the data set is redundant and huge, stochastic gradient descent makes more sense. Since, as opposed to batch learning, stochastic gradient descent changes the weights after computing gradient for every sample. Instead of changing the weights for every sample, we use mini batches after which we perform the weight updates. Stochastic gradient learning is the method of choice, since it is usually much faster than batch learning, leads to better solutions and is advantageous in tracking changes.

3. **Normalization and shifting the inputs**
   When using steepest descent, it usually helps to transform each input component so that it has **zero mean** over the entire training set.

   Scaling the inputs, so that each component of the input vector has unit variance over the whole training set. This helps because if the inputs vary a lot in their magnitude the error surface in which there is high curvature in the direction where input component is low and vice versa. Hence, if we scale the input, the error surface is more circular or the curvature is almost the same in all the directions.

4. **Comparison of various activation functions**
Softmax regression (or multinomial logistic regression) is a generalization of logistic regression to the case where we want to handle multiple classes.
The tanh(z) function is a rescaled version of the sigmoid, and its output range is [1,1][1,1] instead of [0,1][0,1]. Hence, using tanh(z) helps in preserving all the outputs in the range [-1,1] that is the standard deviation of the outputs is still preserved to be unit and the mean is roughly zero. That is when the inputs have mean zero and unit standard deviation.

5. **Initialization of the input weights**
If two hidden units have the same bias and exactly the same incoming and outgoing weights, then they will always get the same gradient. Hence, they would never learn different features. To break this symmetry, we can initialize the weights with small random values.
If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot or undershoot. We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to square root of the fan in. This helps as if the fan in large, the weights are smaller and vice versa. We can also scale the learning rates according to the fan in.

6. **Momentum**
If we imagine a ball on the error surface, the location of the ball in the horizontal plane represents the weight vector. We don't use gradient to change the position of the weight as done in the standard method. We use gradient to change the acceleration of the weights and use velocity to change the position of the weight. The reason that it is different is because the weights have momentum that is they remember the last change in their weights.

What momentum does is that it damps the oscillations in the direction of high curvature.
$v(t) = \alpha v(t-1) - \varepsilon \partial E(t)/\partial (w)$
$\Delta w(t) = v(t)$

Here alpha is a metaparameter with value 0.9. It's better to have smaller momentum in the beginning as the gradients are very big. Then smoothly raise the momentum in the later stages when weights are stuck in the ravines and momentum can help them out. Also, small learning rate and large momentum helps in a bigger learning rate than we would have had with the learning rate alone, also if we use large learning rate then there would be big oscillations.

Nesterov momentum: The standard momentum method first computes the gradient at the current location and then takes a big jump in the direction of the updated accumulated gradient. While the Nesterov method, first make a big jump in the direction of the accumulated gradient and then measure the gradient where the weights end up and make a correction.

$v(t) = \alpha v(t\ \ 1) \quad \varepsilon \partial E(w(t-1) + \alpha v(t\ \ 1)\ )/\ \partial (w)$

4.2 Methods used

We used the initial learning rate 0.0001 (for 3(e)). We also annealed the learning rate with every epoch by using the 1/t decay which has the mathematical form $\alpha=\alpha_0/(1+kt)$ where α_0, k are hyperparameters and t is the iteration number. We used α_0 as 0.001 and k as 1/300.

We used mini batch gradient descent with batches of size 128.

We used Nesterov momentum method to enhance the learning speed with the alpha metaparameter as 0.9. We used 1.7159tanh((2/3)*x) as the sigmoid function in our hidden layer.

4.3 Results and discussion

(a) Shuffle the examples, and then use stochastic gradient descent. For the purposes of this class, use "minibatches", in which you compute the total gradient over, say, 128 patterns, take the average weight change (i.e., divide by 128), and then change the weights by that amount.

We shuffled the examples and then performed stochastic gradient descent with a mini batch of size 128.

(b) Stick with the normalization you have already done in the previous part, which is a little different than what he describes in Section 4.3 - that can be expensive to compute over a whole data set.

We divided the input units by 255.0 and subtract the mean from the units.

(c) For the output layer, use the usual softmax activation function. However, for the hidden layer(s), use the sigmoid in Section 4.4. Note that you will need to derive the slope of that sigmoid to use when computing deltas.

We used softmax activation function for the output layer and 1.7159tanh(2x/3) for the hidden layers.

(d) Initialize the input weights to each unit using a distribution with 0 mean and standard deviation 1/sqrt(fan-in), where the fan-in is the number of inputs to the unit.

We initialized the inputs weights to each unit with 0 mean and standard deviation 1/sqrt(fan-in)

(e) Use momentum.

We used Nesterov momentum to enhance the learning speed with the momentum parameter alpha as 0.9.

(f) Comment on the change in performance, which has hopefully improved, at least in terms of learning speed.

After applying all the above optimizations the network learns the data quite quickly and reaches accuracy of ~92% on the test set in the first epoch itself. Adding the Nesterov momentum and the tangential sigmoid for the hidden units had the most prominent effect on the learning speed of the network.

The analysis and advantages of each of the methods employed has been done in the introduction section of this section.

The performance

Learning	**Train accuracy**	**Test accuracy**	**Hold set accuracy**	**Cross entropy error on test**	**Cross entropy error on hold**	**No. of epochs**
Stochastic Learning with the optimizations in 4)	99.608%	96.7%	96.4%	0.242161175527	0.239802522738	15
Batch Learning as done in 3)	97.868%	96.46%	96.84%	0.251247016989	0.289871554239	1000

Batch Learning without any optimizations/tricks or the trade

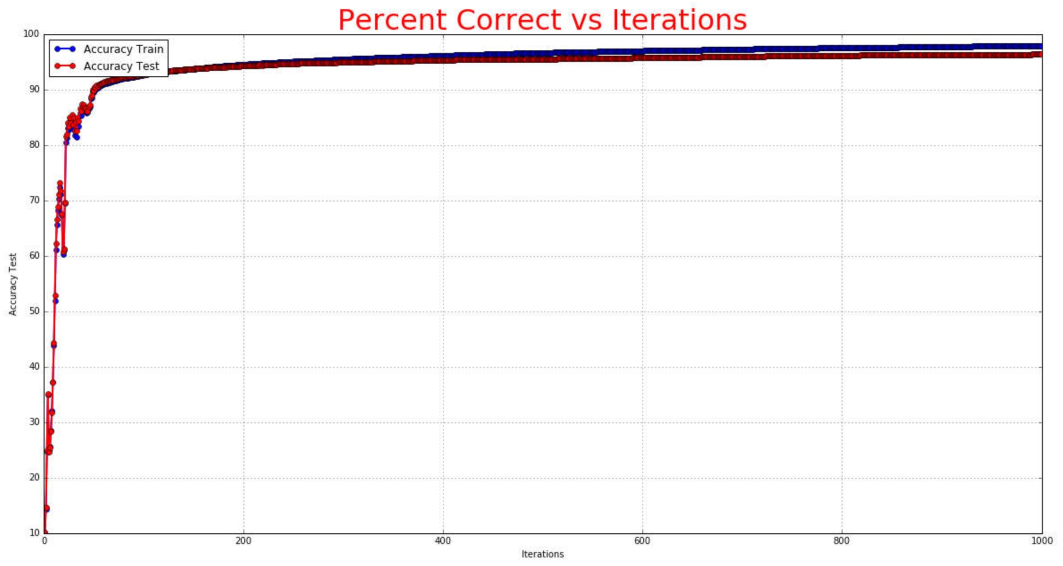

Fig. 4.1 Plot for Batch Learning without any optimizations/tricks of the trade

Mini batch stochastic gradient descent with optimizations

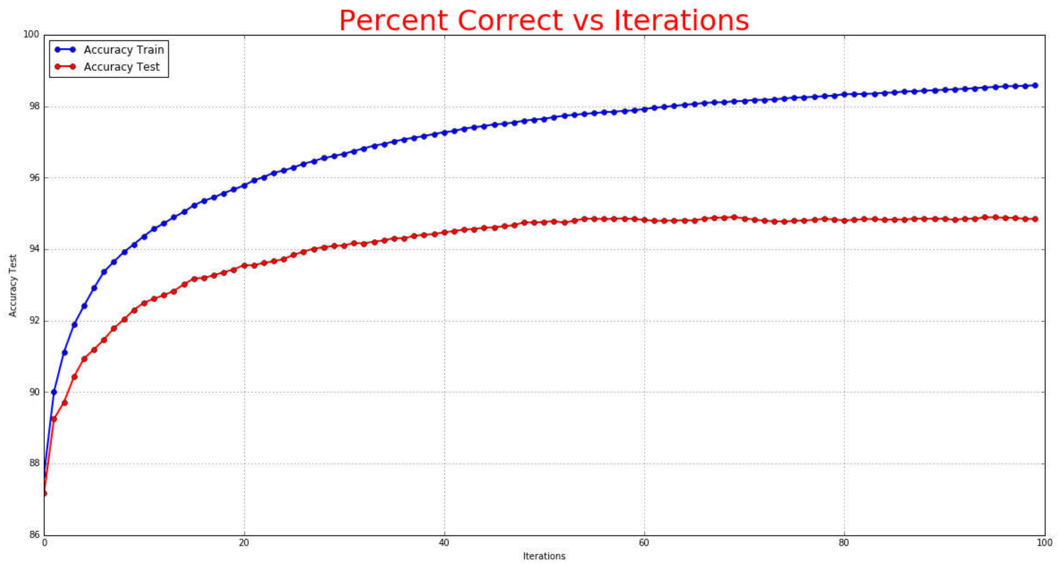

Fig. 4.2 Plot for Mini batch stochastic gradient descent with optimizations

5 Experiment with Network Topology.

5.1 Introduction

Changing the network topologies can affect the neural network in many ways. We have introduced back propagation and the various optimizations in gradient descent in the last two sections.

5.2 Methods used

We tried experimenting with different sizes of the hidden layers.

5.3 Results and discussion

(a) Try halving and doubling the number of hidden units. What do you observe if the number of hidden units is too small? What if the number is too large?

We halved the number of units from 300 to 150, and we got the following results:

Epochs: 10

Accuracy Train:98.434%

Accuracy Test:96.07%

Accuracy Hold:95.47%

Cross entropy error Test:0.221304964085

Cross entropy error Hold:0.254461676577

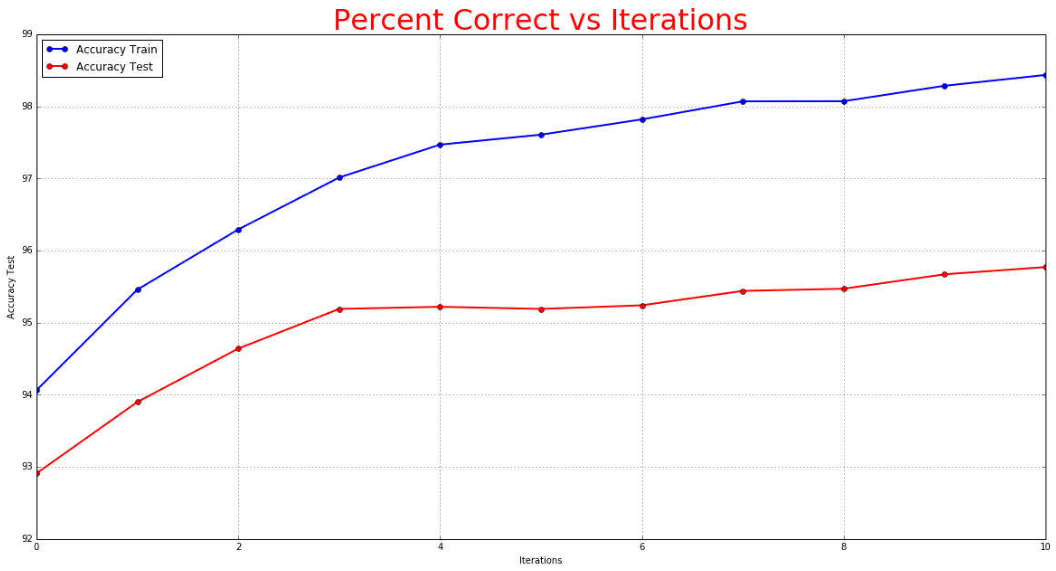

Fig. 5.1 Results obtained by using 150 hidden units in the hidden layer.

We then doubled the number of units from 300 to 600 and got the following results:

Epoch: 10

Accuracy Train:99.188%

Accuracy Test:96.58%

Accuracy Hold:96.61%

cross entropy error Test:0.330033445354

cross entropy error Hold:0.368092995967

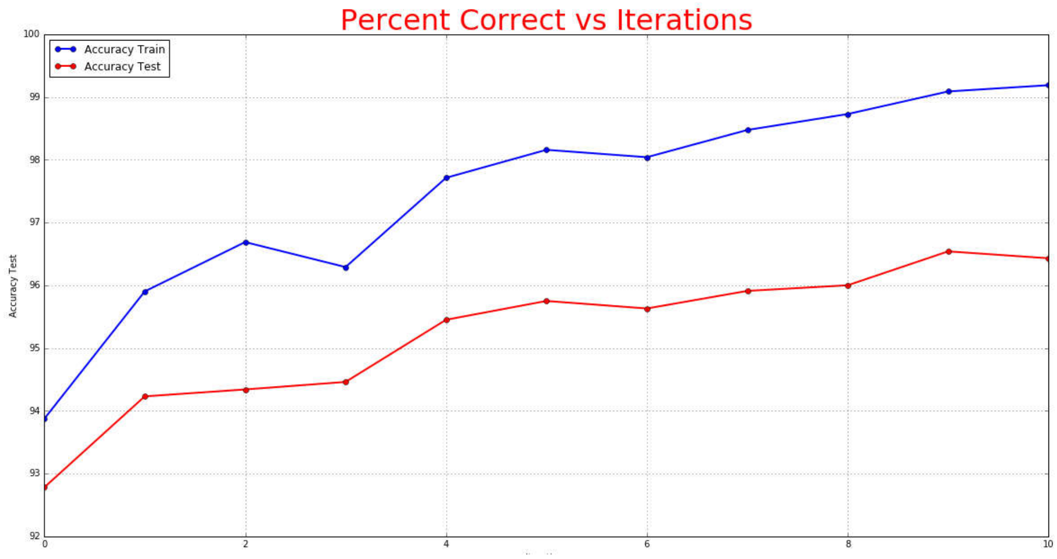

Fig. 5.2 Results obtained by using 600 hidden units in the hidden layer.

The results were not much different and we were able to get similar maximum accuracy on the test and hold sets. But the time taken by the network to saturate decreases with the decreases in the hidden nodes and vice versa.

We also tried increasing the hidden units to 1000 and found that the results are almost the same as that of 300 hidden units but the network takes much more time to train because of the increase in the number of hidden units. Also, if we have a large excess of nodes, the network becomes a memory bank that can recall the training set to perfection, and there is a chance of overfitting the data.

Epoch: 7

Accuracy Train:98.4375%

Accuracy Test:96.67%

Accuracy Hold:96.5%

cross entropy error Test:0.551718559852

cross entropy error Hold:0.555136747866

We also tried training the network with 5 units and the performance of the network fell enormously. This can be attributed to **underfitting** that is the network is not able to learn the training features well if the number of hidden units is too less. The network's accuracy just reaches 25% in the first epoch even on the training set. The maximum accuracy on the training set is also just 78.125% which clearly shows underfitting.

Epoch: 44

Accuracy Train:78.125%

Accuracy test:76.26%

Accuracy Hold:75.06%

cross entropy error Test:0.96126272949

cross entropy error Hold:0.984335224794

So, it's better to use an optimal number of units that is not too high or too small.

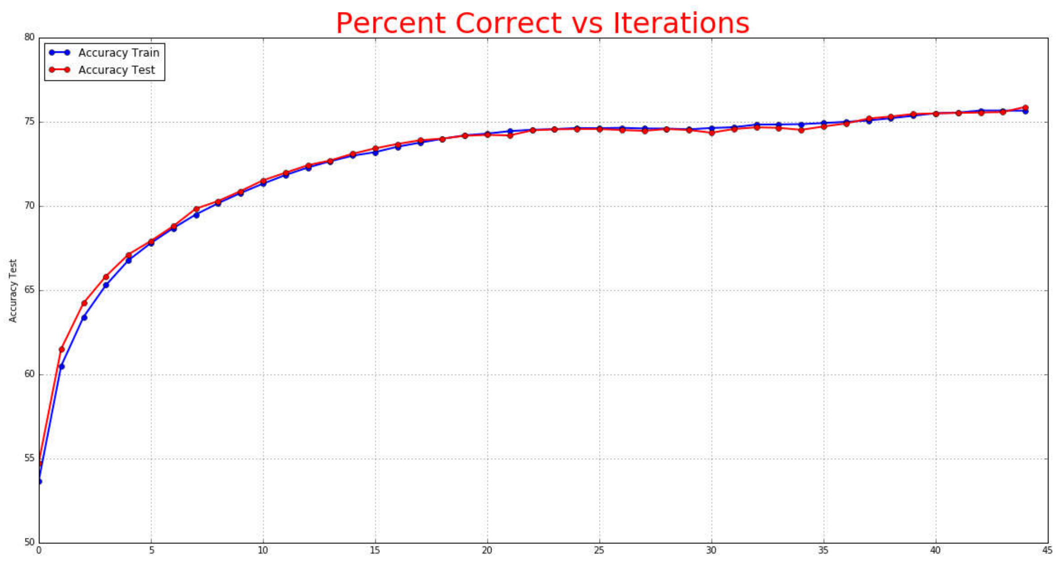

Fig. 5.3 Results obtained by using 5 hidden units in the hidden layer.

(b) Change the number of hidden layers. Use two hidden layers instead of one. Create a new architecture that uses two hidden layers of equal size and has approximately the same number of parameters, as the previous network with one hidden layer. By that, we mean it should have roughly the same total number of weights and biases. Report training and testing accuracy vs. number of training iterations of gradient descent.

We implemented 2 hidden layers, both with 232 hidden units each so that the number of parameters in the previous network and this network is approximately the same. The performance is almost the same, though the system learning time increased. This can be attributed to the fact that as the number of layers is increased the system has become more complex and would need different optimizations to run faster and efficiently.

Epochs: 30

Accuracy Train:98.4375%

Accuracy test:95.43%

Accuracy hold:95.48%

cross entropy error Test:0.23404419687

cross entropy error Hold:0.234415321943

We observed that the performance is almost the same. This is because the MNIST dataset is a simple dataset. Hence only one hidden layer is sufficient enough to correctly classify the handwritten digits in the MNIST dataset. When we add another hidden layer, then the extra hidden units are not learning extra features, therefore the performance remains almost the same.

But, if the dataset had been more complex, then adding a new hidden layer should have improved the performance. This is because in this case, the extra hidden units would have learned some extra features which would finally help in improving the performance of the classifier.

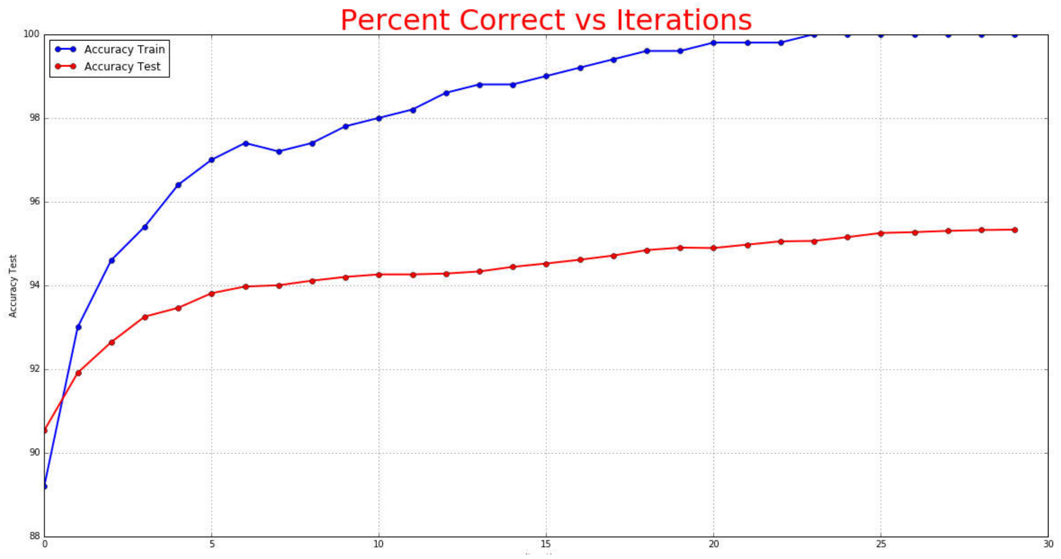

Fig. 5.4 Results obtained by using 2 hidden layers having 232 hidden units (nodes) each.

6 Summary

We performed 10-way classification on the MNIST dataset using multi-layer neural networks (feedforward propagation and backpropagation algorithms) achieving an accuracy of **96.7%** on test set and **99.608%** on training set using the 'tricks of the trade' techniques (discussed in detail in section 4). From this assignment, we learned the feedforward propagation and backpropagation algorithms. We also learned various 'tricks of the trade' techniques such as shuffling the data, data normalization, stochastic gradient descent, Nesterov momentum etc. Finally, we experimented by decreasing and increasing the number of hidden units in the hidden layer. We observed that as the number of hidden units decreases, the accuracy also decreases and when the number of hidden units increases the accuracy also increases. But, when the number of hidden units in the hidden layer is too large (say 1000), then the results are almost same as that of 300 hidden units. Thus, we can say that the accuracy increases with an increase in the number of hidden units, but after a point it becomes constant.

7 Contributions

- This programming assignment was solved by both the team members together. Hence, the contribution of each member is 50%.
- The practice of pair programming was also followed by both the team members.

References

[1] https://en.wikipedia.org/wiki/Backpropagation