CS 391L Machine Learning
Assignment 4
April 2019

Nidhi Kadkol
UT EID: nk9368
Email id: nidhik@cs.utexas.edu

1 Introduction

This project seeks to understand and implement backpropagation by hand in neural networks. Backpropagation is a technique used to train deep neural networks. Artificial Neural networks are used for a variety of prediction tasks. In the case of classification, we give an input to the neural network and it outputs the label of the class it has predicted the input to belong to. In order to train the neural network to give as accurate a prediction as possible on average, we need a method to quantify the error that the neural network makes and accordingly change its decision process to improve its predictions. The technique we use is backpropagation. Essentially it calculates the error or the loss of the network when it makes a prediction. Then that loss is propagated backwards and the parameters of the network are changed to minimize the loss of the network.

We train artificial neural networks with 1, 2, and 3 hidden layers using the technique of backpropagation. We also vary the number of units in each layer. We compare the hinge loss function and the cross entropy loss function on the performance of the neural network. We also compare different activation functions - Rectified Linear Unit (ReLU), Exponential Linear Unit (ELU), tanh, and sigmoid.

Our experiments are done on the MNIST dataset which has 60,000 784x784 images for training and 10,000 test images of the same size.

We have also hand-calculated the gradient updates for 2 weights, $w5$ and $w6$ in the sample neural network as shown below. The calculations are in figures 3 and 4. The network is initialized and trained as shown in figures 1 and 2.

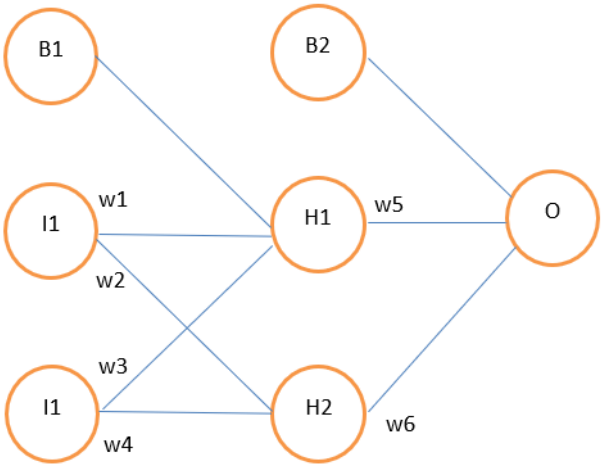

Figure 1: Network

For the given netowrk, the following are initial values:
I1: 0.05, I2: 0.25, B1: 0.45, B2: 0.65, O: 0.01
w1: 0.25, w2: 0.6, w3: 0.75, w4: 0.35, w5: 0.55, w6: 0.65
Activation: Logistic
Loss: Squared Loss
Learning Rate: 0.5

Figure 2: Initializations

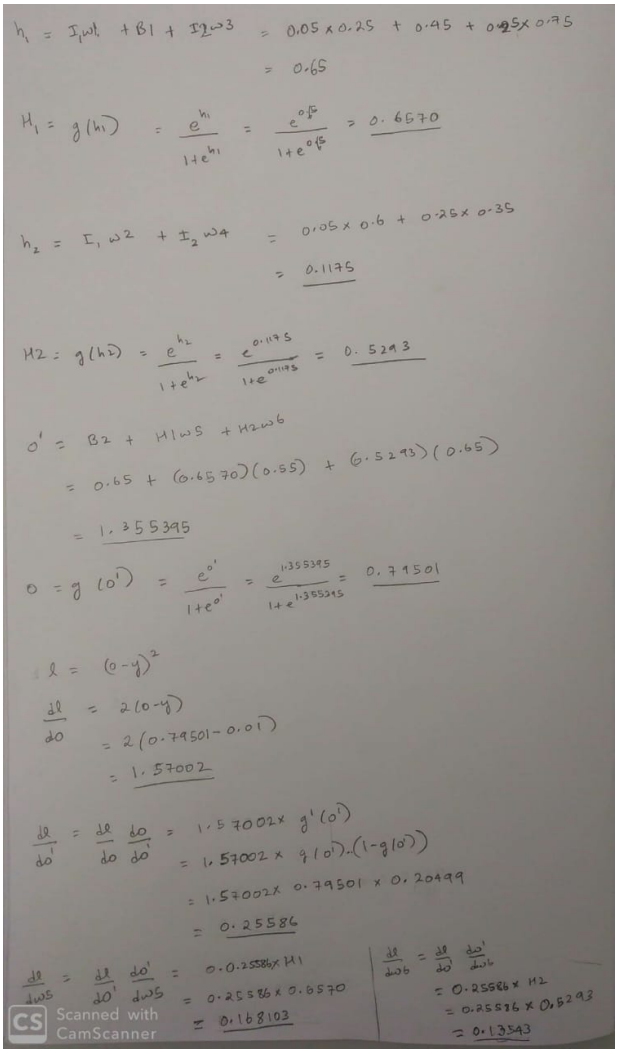

Figure 3: Calculations-1

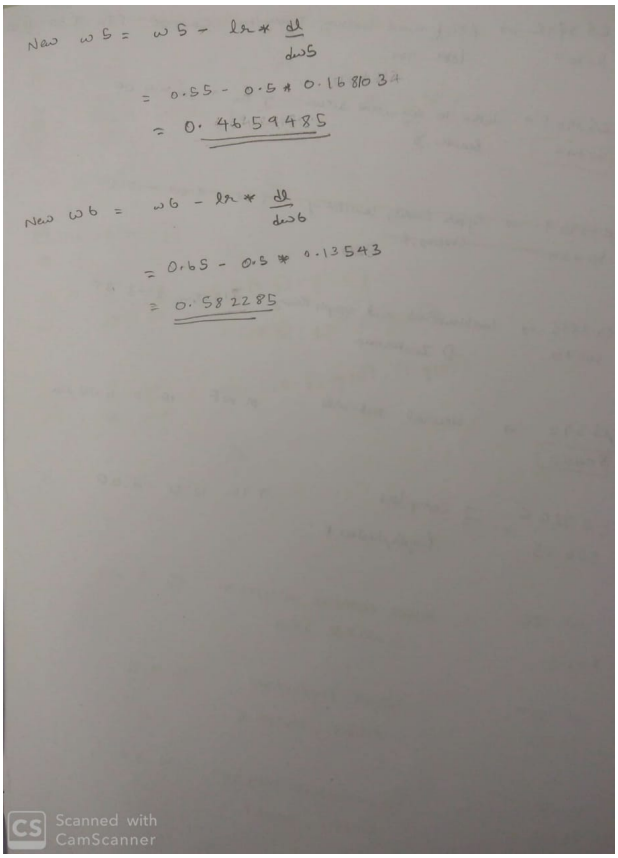

Figure 4: Calculations-2

2 Method

2.1 Gradient Descent

Consider a neural network with one hidden layer as follows:

$$H1_{\text{pre}} = X \cdot W1 + b1$$
$$H1 = g(H1_{\text{pre}})$$
$$S = H1 \cdot W2 + b2$$

The input data is of the form (X, y), where X is an n x m matrix of n training samples in the batch and each sample is an m-dimensional vector. y is an n-dimensional column vector that has the labels for each of the n samples in X. The weights and biases from the input layer to the hidden layer are $W1$ and $b1$, and from the hidden layer to the output layer are $W2$ and $b2$. $g()$ represents the activation function.

Suppose using some loss function we get a loss = l for this network on this input. Now we will show how to us backpropagation to calculate the gradients on each of the parameters $W1, b1, W2, b2$ and also update them.

We first calculate the gradient of the loss function with respect to S, the score matrix. Let this be $M = \frac{dl}{dS}$. Now using chain rule, we propagate the gradients backwards as follows:

$$\frac{dl}{dW2} = \frac{dl}{dS} \cdot \frac{dS}{dW2} = H1^T \cdot M$$
$$\frac{dl}{db2} = \frac{dl}{dS} \cdot \frac{dS}{db2} = M^T \cdot [1, 1, \cdots, n \text{ times}]$$
$$\frac{dl}{dH1} = \frac{dl}{dS} \cdot \frac{dS}{dH1} = M \cdot W2^T$$
$$\frac{dl}{dH1_{\text{pre}}} = \frac{dl}{dH1} \cdot \frac{dH1}{dH1_{\text{pre}}} = \frac{dl}{dH1} \times g'(H1_{\text{pre}})$$
$$\frac{dl}{dW1} = \frac{dl}{dH1_{\text{pre}}} \cdot \frac{dH1_{\text{pre}}}{dW1} = X^T \cdot \frac{dl}{dH1_{\text{pre}}}$$
$$\frac{dl}{db1} = \frac{dl}{dH1_{\text{pre}}} \cdot \frac{dH1_{\text{pre}}}{db1} = (dH1_{\text{pre}})^T \cdot [1, 1, \cdots, n \text{ times}]$$

If there is more than one layer, instead of $H1 \cdot W2 + b2$ being equal to S, it becomes equal to $H2_{\text{pre}}$, and then we continue the same pattern of equations till we get S.

Once we get the gradients of the loss with respect to the parameters, we update them by subtracting their gradient multiplied by a small positive constant called the learning rate. This is known as gradient descent.

2.2 Loss functions

The first loss function we use is the SVM Loss or Hinge Loss. The mathematical formulation is as follows:

$$\sum_{j \neq y_i} \max(0, s_{ij} - s_{iy_i} + 1)$$

where s_{ij} is the element in the i^{th} row and j^{th} column of S, or the score given to class j for input i. For a minibatch, we take the average of the loss of the batch. The gradient of the loss with respect to S is a matrix of the same dimensions as S, with 1s in the positions where $s_{ij} > 0$, $-\sum_{j:j \neq y_i, s_{ij} > 0} 1$ in positions s_{iy_i}, and 0s in the rest of the positions. It is divided by the number of samples in the batch.

The second loss function we use is the Cross Entropy (CE) Loss. Here, we first convert each score to a probability. We do this by updating each s_{ij} to

$$p_{ij} = \frac{e^{s_{ij}}}{\sum_j e^{s_{ij}}}$$

Then the Cross Entropy loss for an example is $-\log p_{iy_i}$. For a minibatch, we take the average of the loss of the batch. The gradient of the loss with respect to S is the matrix P containing p_{ij} in positions where $j \neq y_i$, and $p_{ij} - 1$ otherwise. It is divided by the number of samples in the batch.

2.3 Activations

The **ReLU** activation function takes an input x and returns $\max(x, 0)$. Its gradient takes an input y and returns 1 if $y \geq 0$ and 0 otherwise.

The **tanh** activation function takes an input x and returns $\tanh(x)$. Its gradient takes an input y and returns $1 - y^2$.

The **sigmoid** activation function takes an input x and returns $e^x/(1 + e^x)$. Its gradient takes an input y and returns $y \cdot (1 - y)$.

The **ELU** activation function takes an input x and returns $\alpha(e^x - 1)$ if $x \leq 0$, and x otherwise. In our implementation, we set α to 1. The gradient of this activation function takes an input y and returns 1 if $y > 0$ and $\alpha + y$ otherwise.

3 Results

3.1 Comparison of activation functions

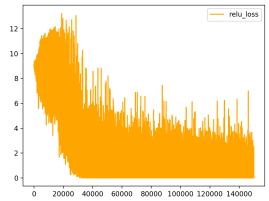

Figure 5: ReLU Loss

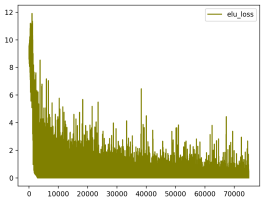

Figure 6: ELU Loss

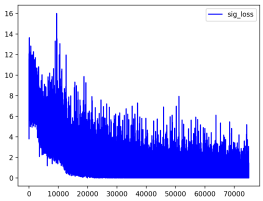

Figure 7: Sigmoid Loss

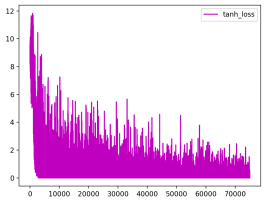

Figure 8: Tanh Loss

We trained a neural network with 2 hidden layers, having 800 units in the first hidden layer and 100 units in the second hidden layer using the hinge loss function. We initialized our network weights from a standard normal distribution scaled by 10^{-3}. We also used mini-batches of size 8 while training. We looked at the results with 4 different activation functions applied on the outputs of the 2 hidden layers with a learning rate of 0.01, trained for 10 epochs. For ReLU, this was too high and we saw exploding gradients and losses which is expected behavior for ReLU as it doesn't constrain the input to within a range. So we used a smaller learning rate of 0.001 for ReLU and trained it for 20 epochs.

Figure 5 shows the graph of the loss of ReLU with a learning rate of 0.01. We see how the loss is unstable and jumps around a lot especially in the first 40k iterations after which is becomes slightly lower. With ReLU, some gradients can be fragile during training and die, which cause a weight update that never activates the neuron again. Thus, it results in a lot of dead neurons which do not respond to variations in error or input. However, due to the simple nature of the ReLU function, we found that it ran much faster than other activations such as sigmoid. From figures 5 to 8, we see that with ELU the loss goes to 0 the fastest of all, which agrees with our theoretical expectation as it doesn't have the vanishing gradient problem that sigmoid and tanh have, and is also doesn't force outputs to be 0 like ReLU, and allows outputs to be negative. We see that the loss for sigmoid is quite erratic, this is because its output isn't zero-centred which makes he gradient updates go too far in different directions. The tanh function gives much better convergence than sigmoid as now the output is 0-centred since its range is between -1 and 1.

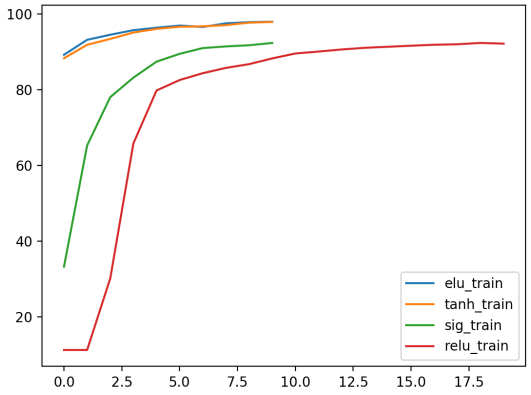

Figure 9: Accuracy vs epochs on training data

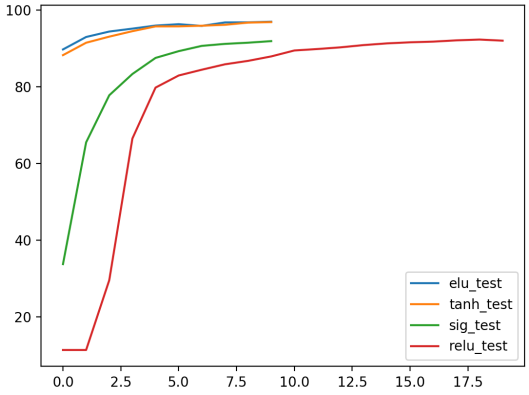

Figure 10: Accuracy vs epochs on test data

From figures 9 and 10, we see that ELU has the best performance. ReLU needs longer to train since we used a lower learning rate. tanh has much better performance than sigmoid activation. From the trend we see that if we train ReLU for longer it will overtake sigmoid and potentially tanh as well. If we train ELU for longer, it will easily be much better than tanh compared to its marginally better performance than tanh now.

3.2 Comparison of Loss functions

Cross Entropy loss interprets the scores as probability distributions. It tries to maximize the probability of the correct class. This leads to a better probabilistic estimation of the scores at the cost of accuracy. On the other hand, hinge loss penalizes predictions not only when they are incorrect, but even when they are correct but not confident. It gives us a high error when the predictions are wrong. It gives some amount of error when the predictions are correct but the score of the correct class is only slightly higher compared to the scores of the incorrect classes. If the score of the correct class is the highest (that is we have a correct prediction) and at the same time it is the highest by a large margin compared to the scores of the incorrect classes, then the error is very small.

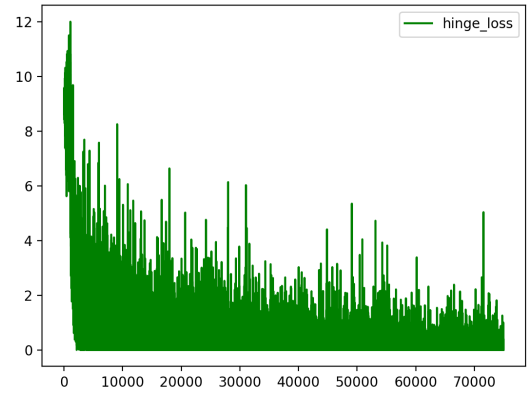

Figure 11: Hinge Loss

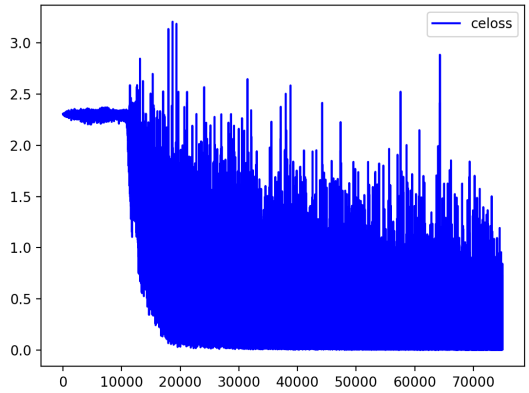

Figure 12: Cross-Entropy Loss

For our experiments, we trained a neural network with 2 hidden layers have 800 units in the first hidden layer and 100 units in the seconds hidden layer. We used a learning rate of 0.01 and initialized our network weights from a standard normal distribution scaled by 10^{-3}. We also used mini-batches of size 8 while training. We trained our networks for 10 epochs with the tanh activation function.

In figures 11 and 12 we see that the hinge loss starts out very high as it penalizes a lot more types of predictions than CE does, leading to much higher losses. However, it goes down to a loss that is lesser than the final CE loss. The CE loss initially oscillates for the first 10k iterations or so before it begins decreasing.

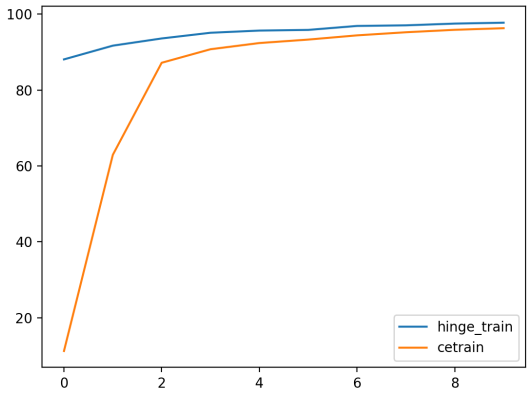

Figure 13: Accuracy vs epochs on training data

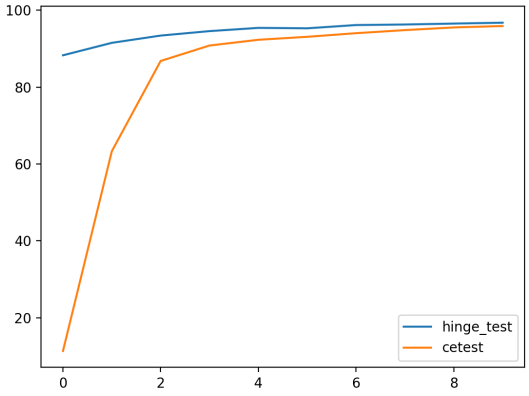

Figure 14: Accuracy vs epochs on test data

In figures 11 and 12, we see that the hinge loss performs better than the cross entropy loss both on train and test data. This agrees with our expectation as explained above in this subsection. We also see very similar behavior on both train and test data, this is because the distribution of the train and test data is similar.

3.3 Performance in terms of network depth

If a network is deeper, then it will be richer in terms of being able to model the data with more parameters. So we expect a deeper network to have better performance.

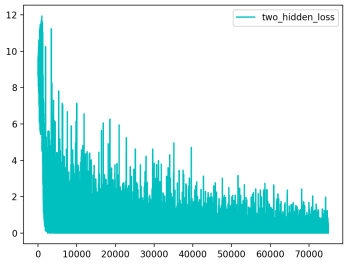

Figure 15: Loss - 1 hidden layer

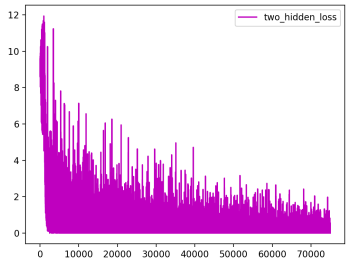

Figure 16: Loss - 2 hidden layers

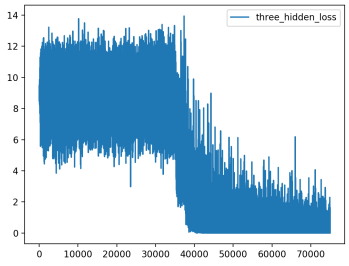

Figure 17: Loss - 3 hidden layers

We used a learning rate of 0.01 and initialized our network weights from a standard normal distribution scaled by 10^{-3} in our experiments. We also used mini-batches of size 8 while training. We used the tanh

activation function coupled with hinge loss. Our 1 hidden layer network had 100 units in the hidden layer. The 2 hidden layer network had 800 units in the first hidden layer and 100 hidden units in the second hidden layer. The 3 hidden layer neural network had 600 hidden units in the first hidden layer, 800 hidden units in the second hidden layer, and 100 hidden units in the third hidden layer.

In figures 15 to 17 we see how the loss function decreases for networks with 1 to 3 hidden layers. The 3-hidden layer doesn't have the loss jump up and down as widely as the 1 hidden layer and 2 hidden layer networks do in the first half of training. At the end of 10 epochs, the loss of the 3 hidden layer network is more than that of the 1 and 2 hidden layer network, also the graph indicates that it has not converged yet. This shows that deeper networks need more time for training.

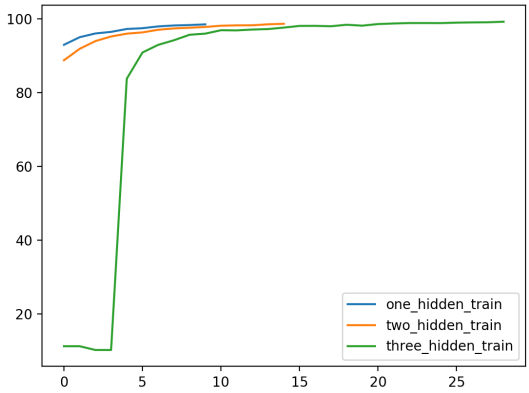

Figure 18: Accuracy vs epochs on training data

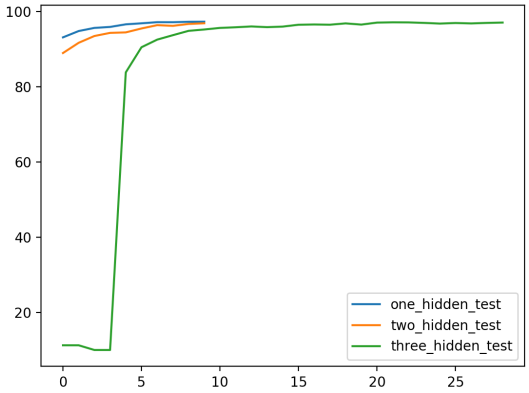

Figure 19: Accuracy vs epochs on test data

In figures 18 and 19 we see that though deeper networks need more time to train, their performance ends up being better than shallower networks.

3.4 Performance in terms of number of units

As with deeper networks, we reason that having more units in a layer of a network would allow more flexibility to model and predict the data, and hence lead to better performance. Also with more units we expect that we will require longer training for convergence as there are more parameters which need to be updated during gradient descent.

For this section of experiments, we train a neural network with 2 hidden layers using mini-batches of size 8, a learning rate of 0.01 with initialization of network weights from a standard normal distribution scaled by 10^{-3}, and the tanh activation function along with hinge loss.

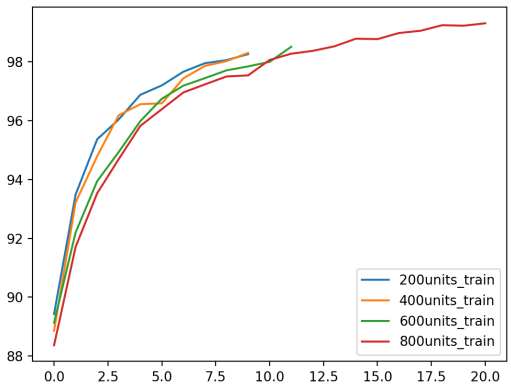

Figure 20: Train accuracy varying number of units in hidden layer 1

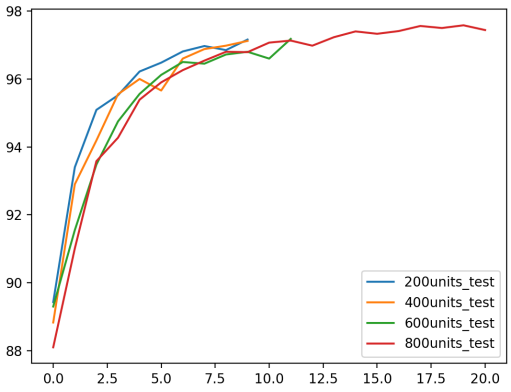

Figure 21: Test accuracy varying number of units in hidden layer 1

Figures 20 and 21 show the results of our first set of experiments where we fixed the number of units in the second hidden layer to be 100 and increased the number of units in the first hidden layer from 200 to 800 in steps of 200. We see that for both train and test data, the accuracy at the end of the last epoch is greater with a larger number of hidden units. The network with 800 hidden units required more epochs to converge than the other networks.

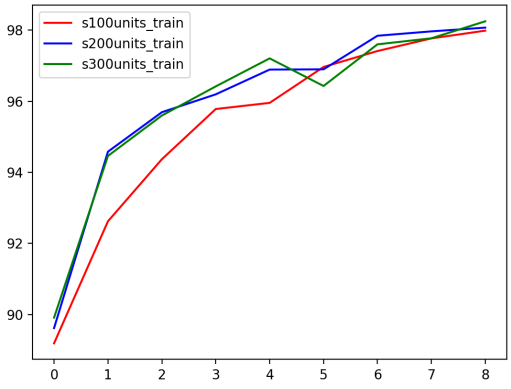

Figure 22: Train accuracy varying number of units in hidden layer 2

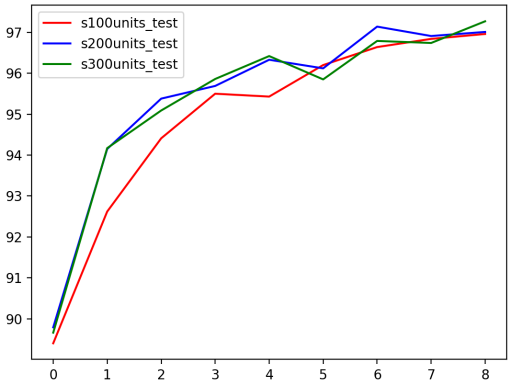

Figure 23: Test accuracy varying number of units in hidden layer 2

Figures 22 and 23 show the results of our first set of experiments where we fixed the number of units in the first hidden layer to be 400 and increased the number of units in the second hidden layer from 100 to 400 in steps of 100. Here we clearly see that for both train and test data, the accuracy is highest with 300 units in the second hidden layer and lowest with 100 units in the same hidden layer.

3.5 Classification

We trained a final model to classify MNIST digits. We used a neural network with 2 hidden layers, having 800 units in the first hidden layer and 100 units in the second hidden layer. We trained it for 25 epochs in mini-batches of size 8 with a learning rate of 0.01 and initialization of network weights from a standard normal distribution scaled by 10^{-3}. We also used the tanh activation function along with hinge loss. Our final train accuracy was **99.31%** and our test accuracy was **97.44%**.

4 Summary

We implemented backpropagation by hand in artificial neural networks using mini-batch gradient descent. We looked at the performance of networks with different activation functions and saw that ELU works the best, with tanh not far behind. We also compared the categorical loss functions, hinge and cross entropy, and saw that hinge loss gives us better performance. We then analyzed the variation of performance with depth and number of units in the hidden layers and saw that the performance is better with deeper networks having more hidden units. Finally, we choose a particular neural network architecture with loss and activation function and trained it to classify images from the MNIST dataset. We got very high accuracies with a 2-layer neural network with a sufficiently high number of hidden units.