EXERCISE 4

STATISTICAL MACHINE LEARNING

Giray Salgir and Gökberk Gül

Tuesday 13th July, 2021

1 Neural Networks

1.a Multi-layer Perceptron

We are given mnist hand-written digit dataset and asked to build a neural network to classify the correct digits. For the architecture of the neural network, we have chosen the input layer to have 784 neurons, since that is the pixel count of the input image. The output layer has 10 neurons since we have total 10 digits to classify. Then, we have added one hidden layer in-between which the neuron number we change and test in order to observe the effects. We didn't have the need for extra hidden layers since we have rather small, black and white dataset and a simple classification problem.

For the activation functions, we have chosen output layer to have softmax function. It is a general application to use softmax function when the problem consists of multi-class classification. Softmax function is defined as

$$\sigma(Z) = \frac{e^Z}{\sum_{j=1}^{K} e^{z_j}} \; for \; Z = (z_1, z_2, ..., z_K)$$

Advantage is that softmax outputs a vector that sums up to 1 which can be interpreted as probability for each class. For the hidden layer, we chose ReLU as the activation function. We have two reasons for that, first is when sigmoid is used, vanishing gradients can be a problem. Second is, ReLU is a frequently used activation function in the recent years. ReLU is defined and has the shape as;

$$f(X) = max(0, X)$$

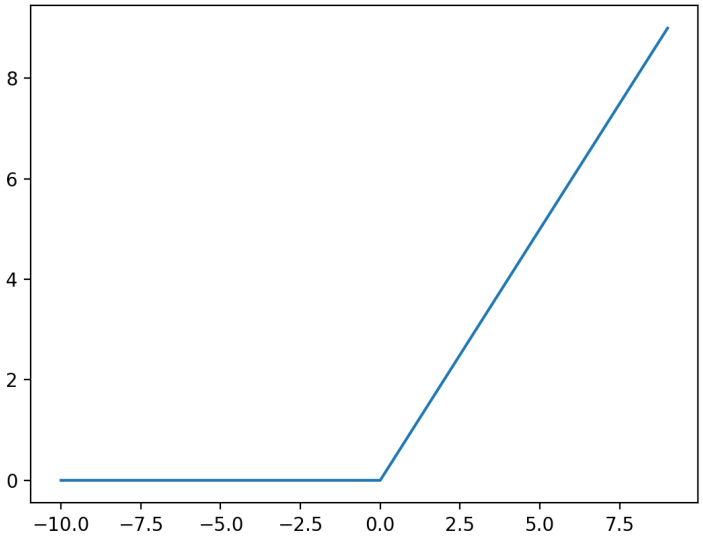

Figure 1: ReLU Shape

For the optimizer, we have chosen batch gradient descend. It is faster than classical gradient descend since all dataset is not used for one iteration, but it is more stable than stochastic gradient descend since a group of data is considered for one iteration instead of one. Batch size is again adjusted to find the optimal one in the experimentation.

Lastly, since the network had problems with learning (overfitting) on the initial tests, we have added a L2 regularization term. The cost function with the regularization term is defined as

$$J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^i, y^i) + \frac{\lambda}{2m}\sum_{l=1}^{L} L(\left|W^i\right|^2)$$

$$\textit{Where L is the cross entropy function } L = \frac{1}{m}\sum_i \sum_k y_k^i log(\hat{y}_k^i)$$

We have loaded the given mnist dataset into numpy arrays. Before the training process, we have shuffled both the training and testing data for faster convergence. The shuffle code is given as;

```
def shuffle(X_train, X_test, y_train, y_test):
    idx_train = np.random.permutation(y_train.shape[0])
    idx_test = np.random.permutation(y_test.shape[0])
    X_train, y_train = X_train[idx_train], y_train[idx_train]
    X_test, y_test = X_test[idx_test], y_test[idx_test]
    return X_train, X_test, y_train, y_test
```

Then we have implemented the Neural Network with the following code

```
def initialize_parameters(layer_dims):
    # Initilizes the weights as random gaussian of mean 0, var 1
    # Bias vector as zeros
    #np.random.seed(1)
    W = []
    b = []
    for i in range(1, len(layer_dims)):
        W.append(np.random.normal(0, 1, [layer_dims[i-1],layer_dims[i]]))
        b.append(np.zeros((1, layer_dims[i])))
    return W, b

def cost(y, y_hat):
    # Cross entropy cost
    return np.squeeze(-np.sum(np.multiply(np.log(y_hat),y))/len(y))

def L2_regularization(la, weight1, weight2):
    # Adds L2 Regularization to the cost
    weight1_loss = 0.5 * la * np.sum(weight1 * weight1)
    weight2_loss = 0.5 * la * np.sum(weight2 * weight2)
    return weight1_loss + weight2_loss

def test(inputs, labels, W, b):
    # Forward propagates the test dataset with given weight/bias and calculates the accuracy
    input_layer = np.dot(inputs, W[0]) + b[0]
    hidden_layer = np.maximum(0,input_layer)
    scores = np.dot(hidden_layer, W[1]) + b[1]
    probs = np.exp(scores)/np.sum(np.exp(scores), axis = 0, keepdims=True)
    acc = float(np.sum(np.argmax(probs, 1) == labels)) / float(len(labels))
    print("Test accuracy: %", acc*100)
    return acc

def train(X, y, layer_dims, learning_rate, batch_size, num_epocs,
                regularization_term, X_test, y_test):
    W, b = initialize_parameters(layer_dims)
    dW, db = initialize_parameters(layer_dims) #Just for shape allocations
    costs = []
    test_errors = []
    m = len(X)

    for epoch in range(num_epocs):
        iteration = 0
        while iteration < len(X):
            X_temp = X[iteration:iteration+batch_size]
```

```
            y_temp = y[iteration:iteration+batch_size]

            # Forward propagation
            Z1 = np.dot(X_temp, W[0]) + b[0]
            A1 = np.maximum(0,Z1) # Relu

            Z2 = np.dot(A1, W[1]) + b[1]
            A2 = np.exp(Z2)/np.sum(np.exp(Z2), axis = 0, keepdims=True) # Softmax
            y_hat = A2

            # Calculate cost
            cost_value = cost(y_temp, y_hat)
            cost_value += L2_regularization(regularization_term, W[0], W[1])
            costs.append(cost_value)

            # Backward propagation
            dyhat = (y_hat - y_temp)/m # derivative of softmax
            dhidden = np.dot(dyhat, W[1].T)
            dhidden[A1 <= 0] = 0 # derivative of relu

            dW[1] = np.dot(A1.T, dyhat)
            dW[1] += regularization_term*W[1] # regularization
            db[1] = np.sum(dyhat, axis=0, keepdims = True)

            dW[0] = np.dot(X_temp.T, dhidden)
            dW[0] += regularization_term*W[0] # regularization
            db[0] = np.sum(dhidden, axis=0, keepdims = True)

            # Update parameters
            for i in range(2):
                W[i] = W[i] - learning_rate*dW[i]
                b[i] = b[i] - learning_rate*db[i]

            if iteration == 0:
                print("=== Epoch:", epoch, "Loss:", cost_value, "===")
                test_errors.append(100*(1-test(X_test, y_test, W, b)))

            iteration += batch_size

    return costs, W, b, test_errors
```

With each experiment, we have saved the cost and error rate for each iteration. We have run 1500 epochs for each iteration since around 1000 were the limit for less than 8 percent error in

test dataset. Our initial parameter values were

```
layer_dims = [784, 20, 10],
batch_size = 10,
num_epochs = 1500,
learning_rate = 0.001,
regularization_coefficient = 0.01

Which resulted;

Loss: 0.3722690799089119 Test accuracy: 92.03187250996015%
```

The plots for these parameters are

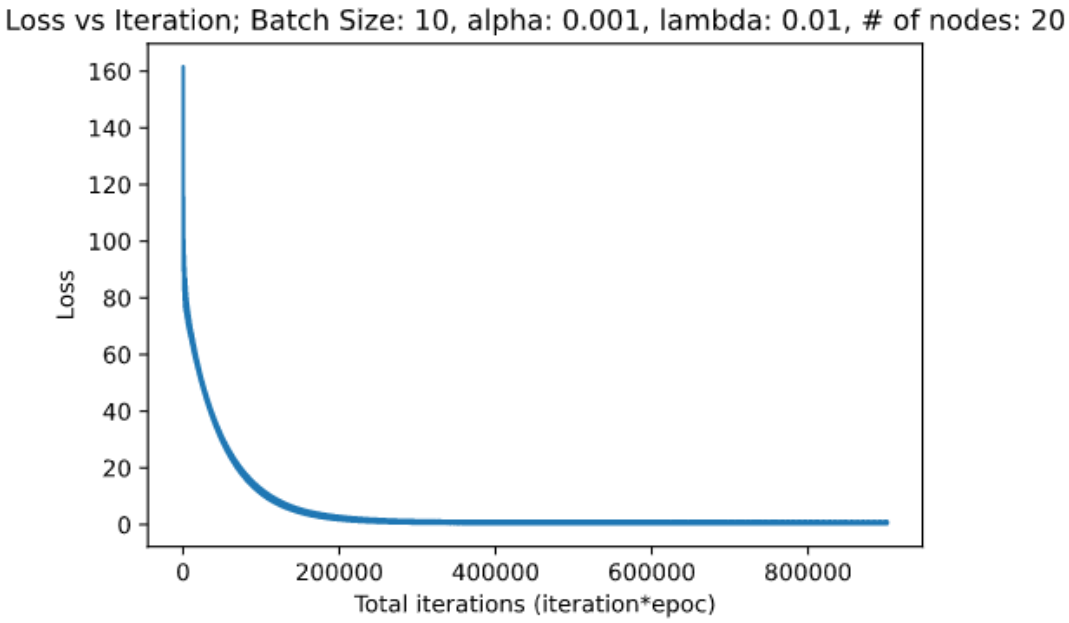

Figure 2: Loss vs Iteration

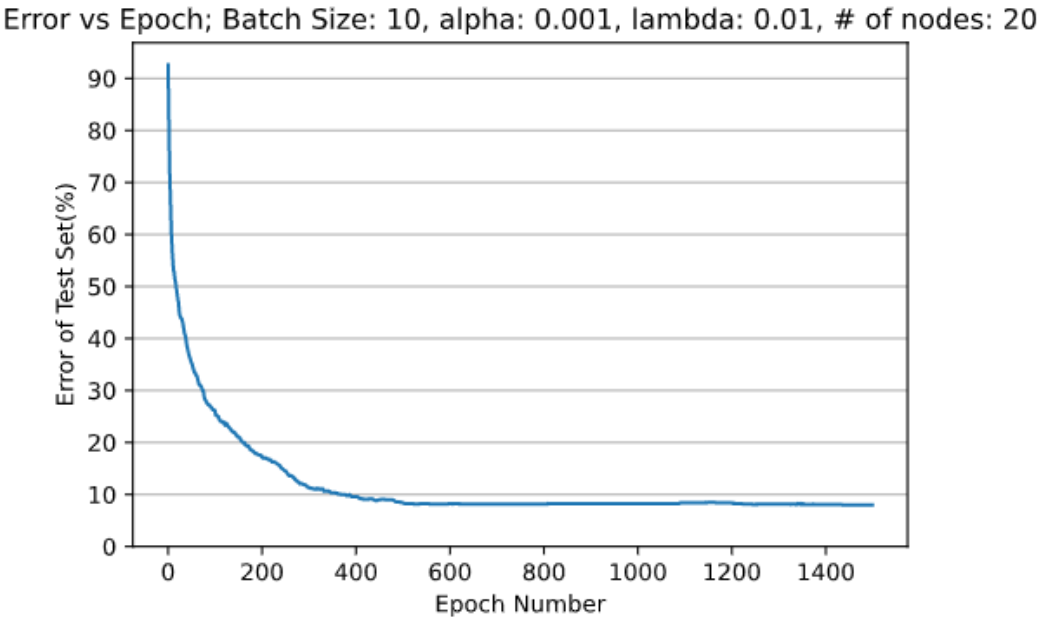

Figure 3: Error vs Epoch

For this case, around 350-400th epoch, %10 error is achieved and around 1200th epoch, %8 error is achieved. Then, we have changed the learning rate, new parameters became;

```
layer_dims = [784, 20, 10],
batch_size = 10,
num_epochs = 1500,
learning_rate = 0.001,
regularization_coefficient = 0.01

Which resulted;

Loss: 0.3371688088713405 Test accuracy: 92.13147410358566%
```

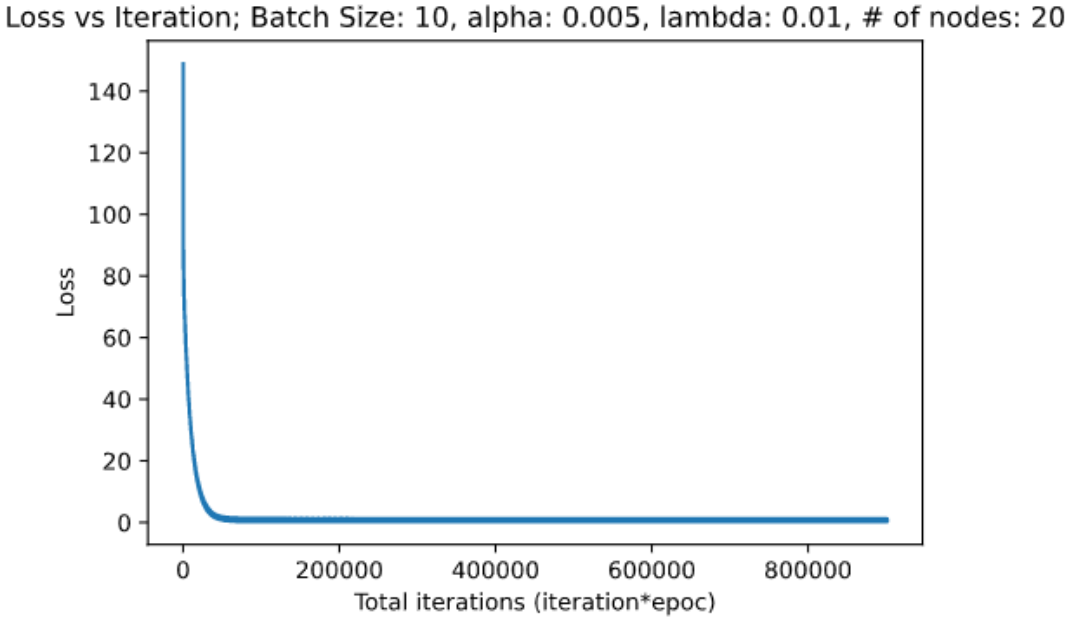

Figure 4: Loss vs Iteration

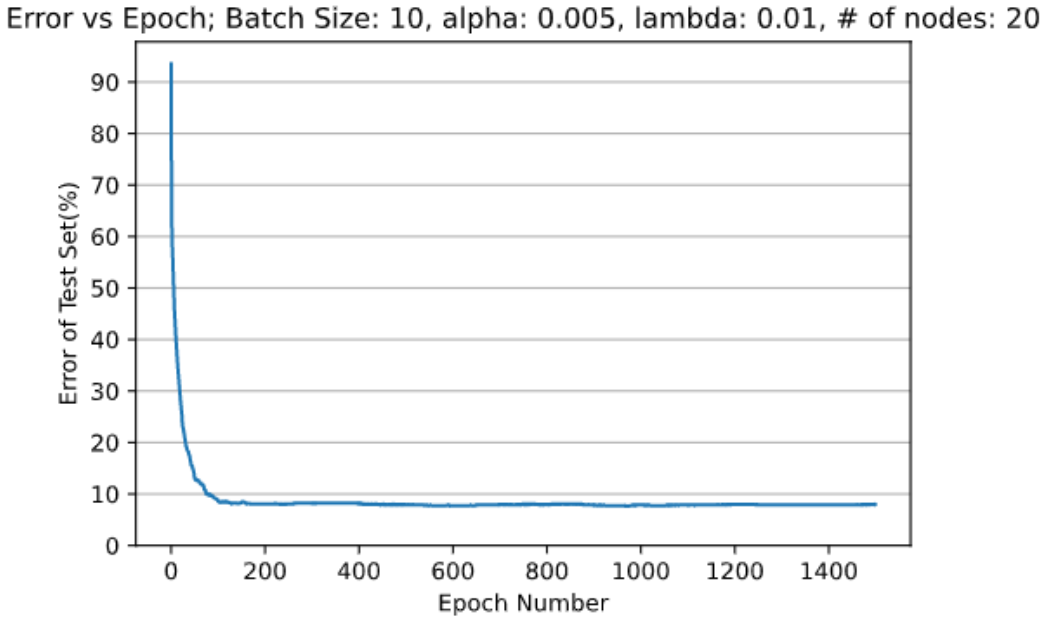

Figure 5: Error vs Epoch

Last results seems to be almost identical to the previous case, however it converged much faster. So, using this learning rate seems to be more logical. With keeping the learning rate 0.005, we have then changed the number of neurons in the hidden layer from 20 to 40.

```
layer_dims = [784, 40, 10],
batch_size = 10,
num_epochs = 1500,
learning_rate = 0.005,
regularization_coefficient = 0.01

Which resulted;

Loss: 0.5570705451965956 Test accuracy: 93.12749003984064%
```

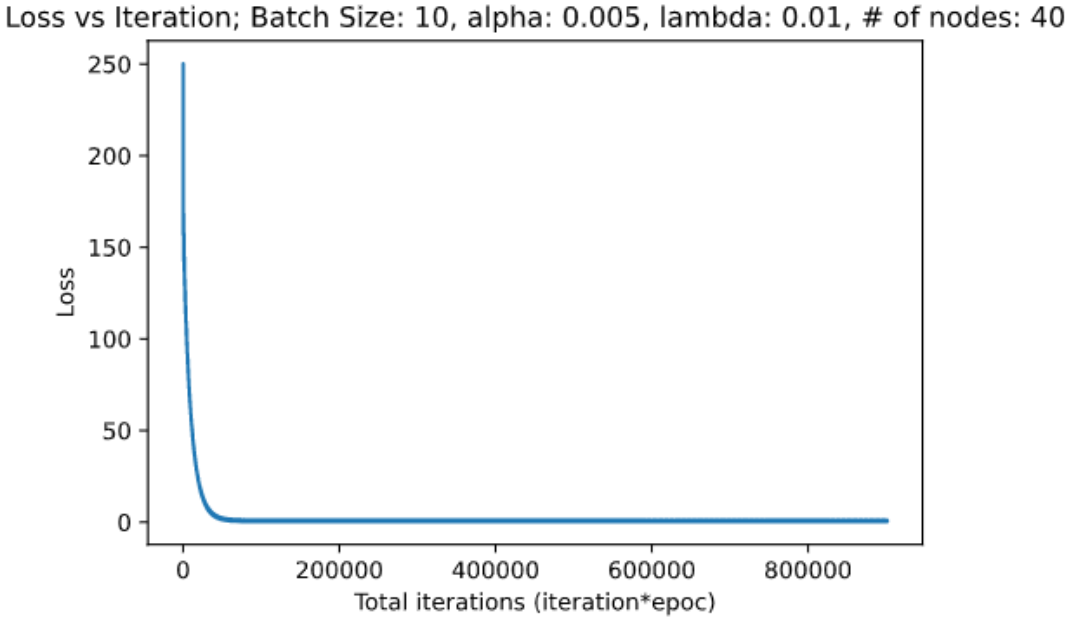

Figure 6: Loss vs Iteration

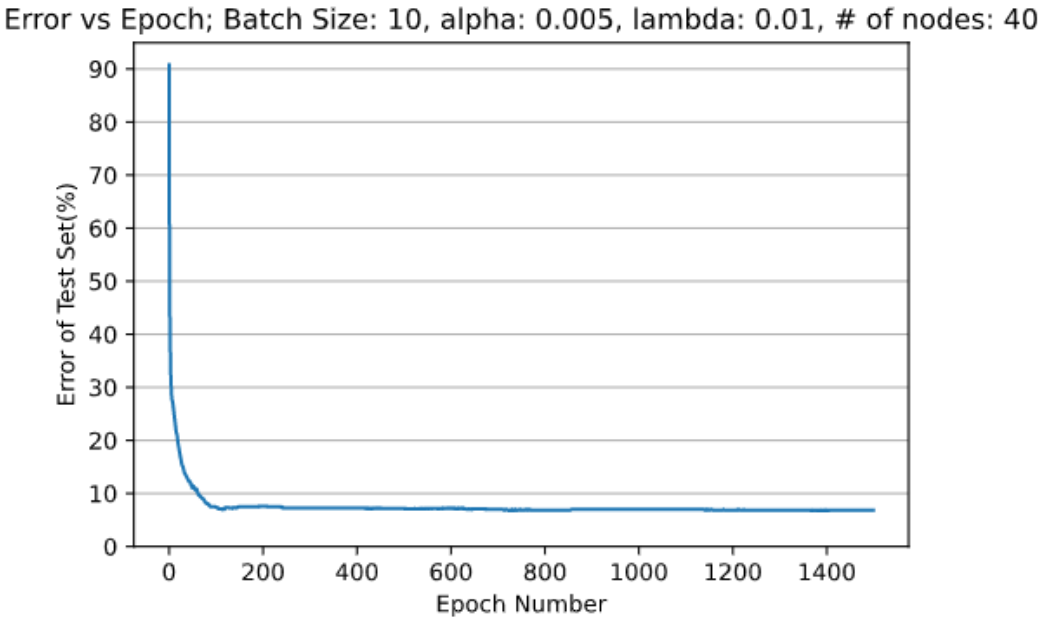

Figure 7: Error vs Epoch

In this case, it converged around same epoch number, however the final result performs better on test set. One problem with increasing the number of nodes is increasing rate of complexity which also increases the computational time. Then, while keeping the hidden layer neuron number constant, we have changed the regularization term from 0.01 to 0.005.

```
layer_dims = [784, 40, 10],
batch_size = 10,
num_epochs = 1500,
learning_rate = 0.005,
regularization_coefficient = 0.005

Which resulted;

Loss: 0.23824896110498722 Test accuracy: 93.42629482071713%
```

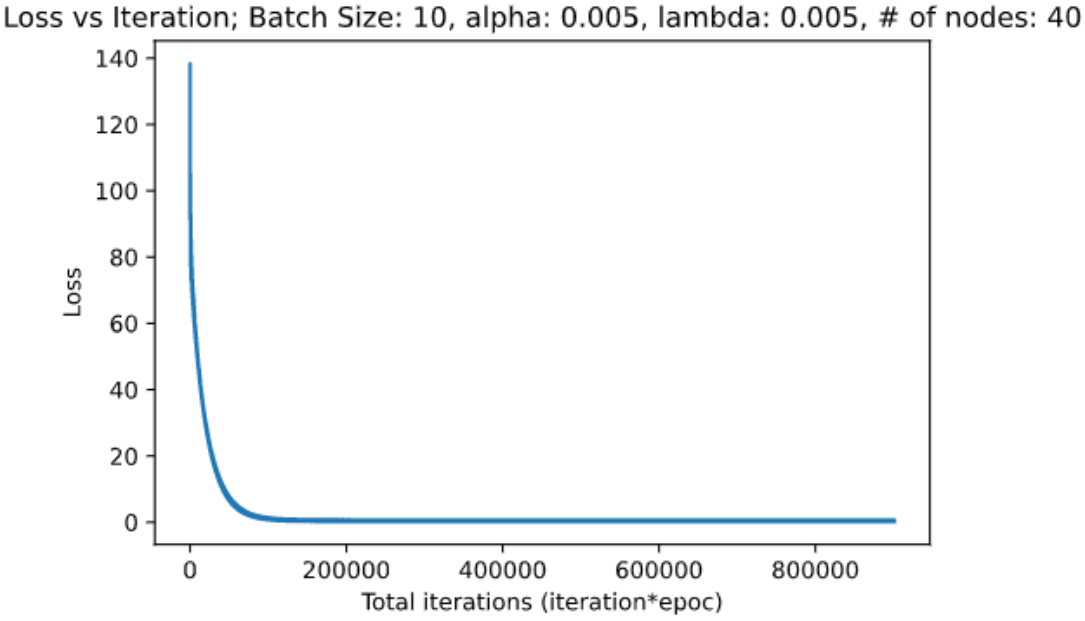

Figure 8: Loss vs Iteration

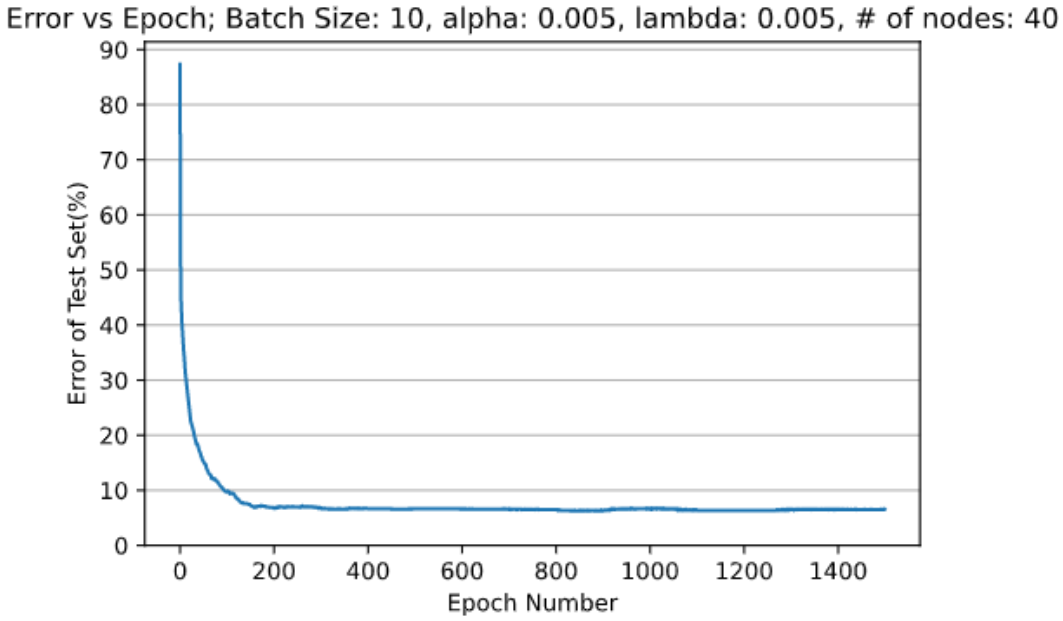

Figure 9: Error vs Epoch

Decreasing the regularization constant slightly increased the convergence time, however the final test set result is better than before. While keeping the regularization constant 0.005, we have then increased the batch size from 10 to 50.

```
layer_dims = [784, 40, 10],
batch_size = 50,
num_epochs = 1500,
learning_rate = 0.005,
regularization_coefficient = 0.005

Which resulted;

Loss: 0.3102510621263144  Test accuracy: 92.92828685258964%
```

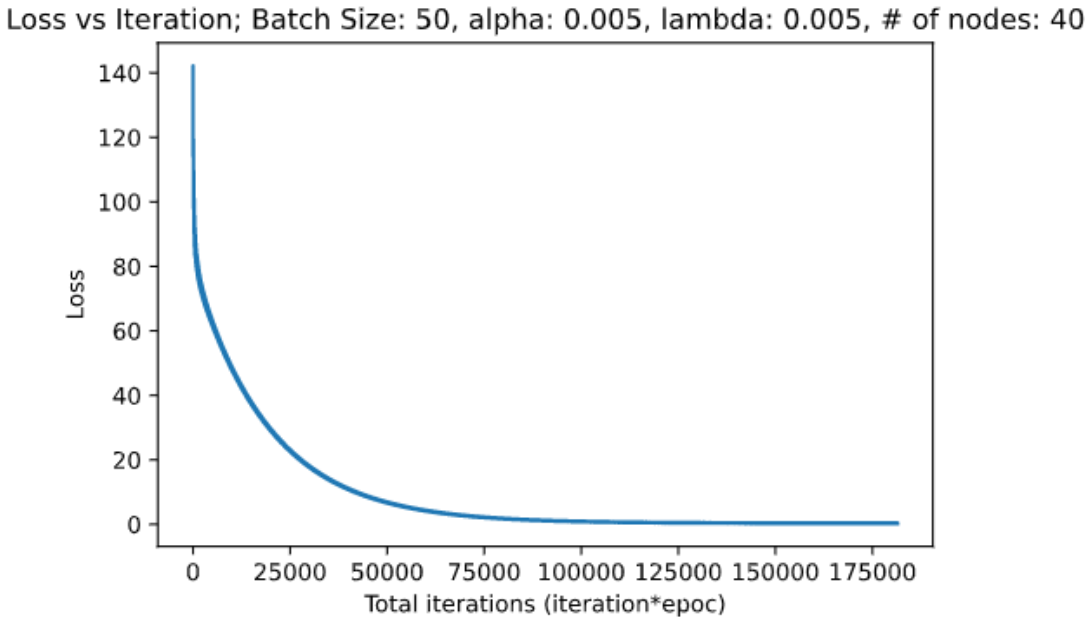

Figure 10: Loss vs Iteration

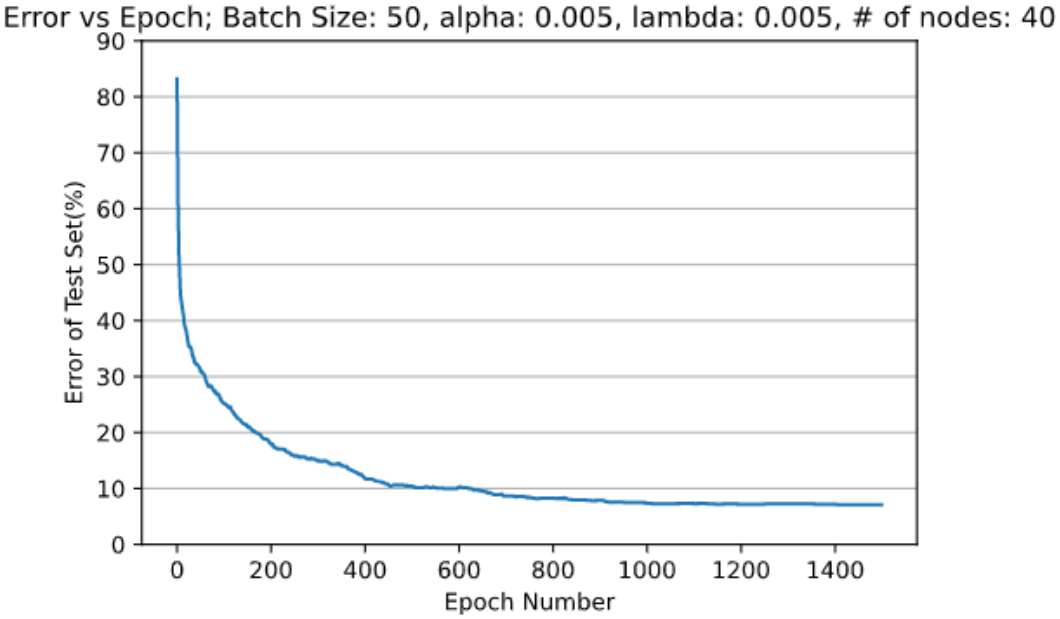

Figure 11: Error vs Epoch

Increasing the batch term resulted with a slightly worse test set accuracy, and has increased the epochs it needs to calculate in order to converge. However, one major advantage is, decreasing the number of iterations per epoch has resulted with a much faster epoch calculation time. So, one work-around for a better test case would be increasing the number of epochs it calculates.

1.b Deep Learning

A neural network is an type of algorithms that are modelled according to the human brain. In a simple form and Artificial Neural Network has 3 layers; one input, one hidden and one output layer, however it can have more than one hidden layer. This artificial neurons make the core computational unit that focuses on uncovering the underlying connections in the given dataset.

A deep neural network, while being similar to the classical neural networks, has one big difference. That is, instead of teaching computers to learn from data, the computer trains itself to learn from data. It consists of several hidden layers which can extract and transform the features in the data. One of the big differences is that deep neural networks can have different architectures than classical neural networks. For example, Convolutional Neural Network works well on image processing, while Recurrent Neural Network works well on time-dependent data such as audio processing. A deep neural network performs better than classical neural networks under optimal conditions.

However, from a theoretical perspective, a deep neural network is not necessarily superior to the classical one. A major drawback is its complexity. It takes much higher time to train. Since there are a lot of hyper-parameters, layers, activation functions etc. it is much harder to design

and control the model. Also, it requires larger dataset to train. So, one should choose whether to use Deep or Classical Neural Network model for the problem at hand, considering all these circumstances.

2 Support Vector Machines

2.a Definition

The aim of Support Vector Machines is to seperate two different classes using an N-dimensional hyperplane where N is the feature number.

Advantages: Unlike other machine learning methods that is discussed in the lecture, SVM can handle non-linear data efficiently using Kernel trick. Also, since it maximizes the margin, the resulting model is far from over-fit.

2.b Quadratic Programming

We can formulize the constrained optimation problem using the formulas in the lecture slides.

$\text{argmin}_{w,b}\, 1/2 * ||w||^2$ such that $y_i * (w^T * x_i + b) - 1 \geq 0$

where w is the parameter vector, x_i s are the input data and y_i is the output of the corresponding datas.

2.c Slack Variables

Slack variables are used for datasets whose classes can not be sepereated linearly using a hyperplane. Slack variables let us called ϵ to allow small violations and thus obtaining a better performance in the modeling. Using the slack variables our constrained optimisation problem turns out to be the following;

$\text{argmin}_{w,b,\epsilon}\, 1/2 * ||w||^2 + C * \sum_{i=1}^{N} \epsilon$

such that

$y_i * (w^T * x_i + b) - 1 + \epsilon \geq 0$

$\epsilon \geq 0$

2.d Slack Variables

In order to solve the optimization problem, we will use the Lagrangian multipliers with variablesm,n which are vector quantities. Our langrangian function becomes;

$L(w, b, m, n) = 1/2 * ||w||^2 + C * \sum_{i=1}^{N} \epsilon - \sum_{i=1}^{N} m_i * [y_i * (w^T * x_i + b) - 1] - \sum_{i=1}^{N} n_i * \epsilon$

We will now take the partial derivative of L with respect to w,b and ϵ

$\frac{\partial L}{\partial w} = 0 -> w = \sum_{i=1}^{N} m_i * y_i * x_i$ $\frac{\partial L}{\partial b} = 0 -> \sum_{i=1}^{N} m_i * y_i = 0$ $\frac{\partial L}{\partial \epsilon} = 0 -> C - m_i - n_i = 0 -> m_i = C - n_i$

Now we will plug these into our equation and following result is obtained;

$L = \frac{-1}{2} * \sum_{i=1}^{N} \sum_{j=1}^{N} m_i * m_j * y_i * y_j * x_i^T * x_j + \sum_{i=1}^{N} m_i$

2.e The Dual Problem

In the previous section we found the dual problem to be;

$L = \frac{-1}{2} * \sum_{i=1}^{N} \sum_{j=1}^{N} m_i * m_j * y_i * y_j * x_i^T * x_j + \sum_{i=1}^{N} m_i$

We can observe from these equation that the dual Lagrangian has fewer unknown variables. Thus, it can be solved easily with linear algebra.

2.f Kernel Tricks

The Kernel Trick uses a function to simulate the scalar product of the data as follows;

$K(x_i, x_j) = \theta(x_i)^T * \theta(x_i)$

By using a Kernel function as above, we make the scalar product directly with a non-linear function instead of mapping the datas to higher dimensional space.