Deep Learning 2020 **(Due Date : 13.11.2020)**

Assignment # 1

Instructor: Jakub Tomczak *Name:* Shwetambari Tiwari, *VUNetid:* sti860

1 Working out local derivatives

Question 1 It is given that

$$y_i = \frac{e^{o_i}}{\sum_j e^{o_j}} \quad (\, For Notation : \sum_j e^{o_j} = \sum \,) \tag{1.1}$$

The derivative of e^{o_j} w.r.t. o_j and o_i can be given as, $\frac{\partial \sum e^{o_j}}{\partial o_j} = e^{o_j}$ *and* $\frac{\partial \sum e^{o_j}}{\partial o_i} = e^{o_i}$
When $i = j$, the derivative of e^{o_i} w.r.t. o_j or o_i is $e^{o_i} = e^{o_j}$. Taking the derivative of (1.1) w.r.t. o_i

$$\frac{\partial y_i}{\partial o_i} = \frac{e^{o_i} \sum - e^{o_i} e^{o_i}}{\sum^2} = \frac{e^{o_i}}{\sum}(1 - e^{o_i}) = y_i(1 - y_i) \tag{1.2}$$

When $i \neq j$, the derivative of e^{o_i} w.r.t. o_j is always 0. Taking the derivative of (1.1) w.r.t. o_j,

$$\frac{\partial y_i}{\partial o_j} = \frac{0 - e^{o_j} e^{o_j}}{\sum^2} = -\frac{e^{o_j}}{\sum} \frac{e^{o_i}}{\sum} = -y_j y_i \tag{1.3}$$

Cross entropy loss for a true class is given as, $loss = -\sum log\ y_c$. Hence, for any class, $loss = -\sum\ t_i\ log\ y_i$. The derivative of loss w.r.t. y_i can be given as $\frac{\partial l}{\partial y_i} = -\frac{t_i}{y_i}$

$$\frac{\partial l}{\partial o_i} = -\sum_{j=1}^{C} \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial o_i} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial o_i} - \sum_{i \neq j} \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial o_i} = -t_i(1 - y_i) + \sum_{i \neq j} t_j y_i = -t_i + y_i \sum_j t_j = y_i - t_i \tag{1.4}$$

2 Scalar Backpropagation

Question 2 The neural network architecture was successfully implemented with plain python and math packages. Results obtained after one forward pass are shown in Table 1. The value of cross entropy loss computed after one forward pass is **0.693**. The derivatives w.r.t to loss were inline with the expected values and are shown in Table 2.

k1(output at layer-1)	[2.0, 2.0, 2.0]
h (sigmoid)	[0.8807970, 0.8807970, 0.8807970]
k2(output at layer-2	[-0.8807970, -0.8807970]
y (softmax)	[0.5 0.5]

Table 1: Results with One Forward Pass

dW	[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
db	[0.0, 0.0, 0.0]
dV	[[-0.4403985, 0.4403985], [-0.4403985, 0.4403985], [-0.4403985, 0.4403985]]
dc	[-0.5 0.5]

Table 2: Derivatives obtained with one Back Propagation

Code snippets for forward pass and back propagation are shown in Figure 1. Detailed code is mentioned in the Appendix.

```
transpose_W = list(map(list, zip(*W)))
transpose_V = list(map(list, zip(*V)))

for j in range(len(transpose_W)):
    for i in range(len(X)):
        k1[j] += W[i][j] * X[i]
    k1[j] += b[j]

for i in range(len(k1)):
    h[i] = sigmoid(k1[i])

for j in range(len(transpose_V)):
    for i in range(len(h)):
        k2[j] += V[i][j] * h[i]
    k2[j] += c[j]

y = softmax(k2)
```

(a) Forward Pass

```
dk2 = y - Y

for i in range(len(transpose_W)):
    for j in range(len(k2)):
        dV[i][j] = dk2[j] * h[i]
        dh[i] += dk2[j] * V[i][j]

dc = dk2

for i in range(len(transpose_W)):
    dk1[i] = dh[i] * h[i] * (1 - h[i])

for j in range(len(transpose_W)):
    for i in range(len(X)):
        dW[i][j] = dk1[j] * X[i]
    db[j] = dk1[j]
```

(b) Back Propagation

Figure 1: Code snippets for neural network implementation

W	[2.8684894, -7.6767074, 6.6785715], [-1.6898393, -2.0112816, -7.7109503]
b	[0., 0., 0.]
V	[-7.2348454, 1.1665972], [1.9808355, -7.8209344], [5.7780672, -8.5619845]
c	[0., 0.]

Table 3: Weights and Bias for synthetic dataset

Question 3 The neural network implemented above is fed with synthetic data set and stochastic gradient is implemented. For synthetic data, training and test set is obtained after loading the data. These are numpy arrays and are converted to lists for further implementation. The weights for this network are normally distributed random values with mean 0 and standard deviation 1, these values are hard coded[1] for stability. Biases are assigned 0 float values. The initialized values to weights and biases are shown in Table 3.The target values are one-hot encoded.

In order to train this network, these 60k data points are fed to the network one at a time . At the end of one forward pass, cross entropy loss is computed. The loss is averaged for this epoch. Thereafter, back propagation is done followed by updating the weights of this network. This process is repeated for several epochs. For training purposes, a learning rate of of 1e-2 is chosen, and stochastic gradient descent is run for 50 epochs. Results are shown in Figure 2. Training and test set are seen to be converging after epoch 30.

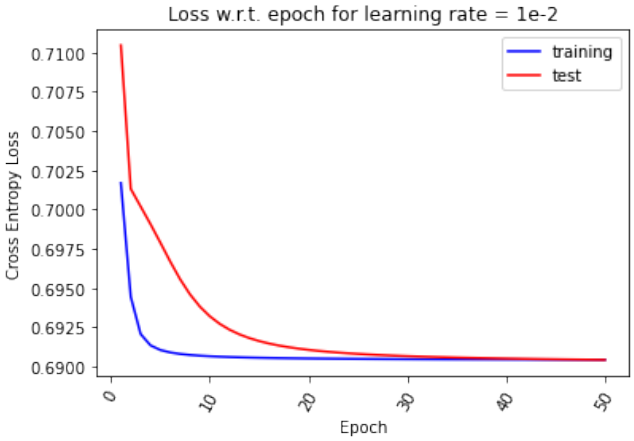


Figure 2: Synthetic Data Set Performance

[1]Generated from https://www.random.org/gaussian-distributions/

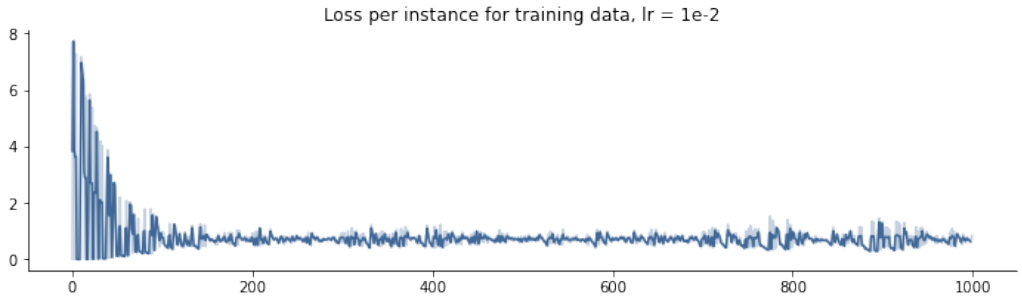


Figure 3: Synthetic Data Training Set- Loss per instance

The loss at epoch 0 for training set is 0.70169 while for the test set it is 0.71046. The loss after 50 epochs is 0.69044 for training set, however, for test set it is 0.69043. With the results obtained, it can said that as the training progresses loss decreases and also that the network is capable of training. Further, hyper parameter tuning can be done. The initialized weights can be tested upon. Batch normalization for this network can also be implemented. Figure 3 shows that the loss stablizes after first 200 instances. Also, training the network over 100 epochs can also show that the loss may decrease further below 0.690 after several epochs.The detailed code can be referred at Appendix.

3 Tensor Backpropagation

Question 4 For mnist dataset, a neural network was implemeted with the architecture $Input(784) - > Linear(784, 300) - > Sigmoid - > Hidden\ Layer(300, 10) - > Softmax$. This dataset had to be normalized. One hot encoding was used to convert the target into categorical type. Stochastic gradient descent is implemeted for this network. Weights and biases are generated using numpy arrays, where weights are randomly generated values, while biases are 0 float values. The input dataset is larger for MNIST dataset which makes this network more complex in comparison to the network implemented for synthetic dataset. Also, MNIST data has 10 classes, which makes prediction more difficult. Figure 4 shows the some code snippet for this implementation. A detailed version of the code is mentioned in the Appendix.

```
# input layer activations becomes sample
params['A0'] = x_train
# input layer to hidden layer 1
params['Z1'] = np.dot(W1.T,params['A0']) + params['b1']
params['A1'] = self.sigmoid(params['Z1'])
# hidden layer 1 to output layer
params['Z2'] = np.dot(params['A1'],params["W2"]) + params['b2']
params['A2'] = self.softmax(params['Z2'])
```

(a) Forward Pass

```
dZ2 = A2-y_train
dW2 = dZ2 * A1.T
db2 = np.sum(dZ2, axis = 1, keepdims = True)
dZ1 = np.dot(dZ2, W2.T) * (A1*(1-A1))
dW1 = dZ1.T * A0.T
dW1 = dW1.T
db1 = np.sum(dZ1, axis = 1, keepdims = True)
```

(b) Back Propagation

```
for iteration in range(self.epochs):
    loss_train = 0
    for x,y in zip(x_train, y_train):
        output = self.forward_pass(x)
        changes_to_w = self.backward_pass(y, output)
        self.update_network_parameters(changes_to_w)
        loss_train = loss_train + self.compute_cost(output,y)
    cost_train = loss_train/m
    train_loss.append(cost_train)
```

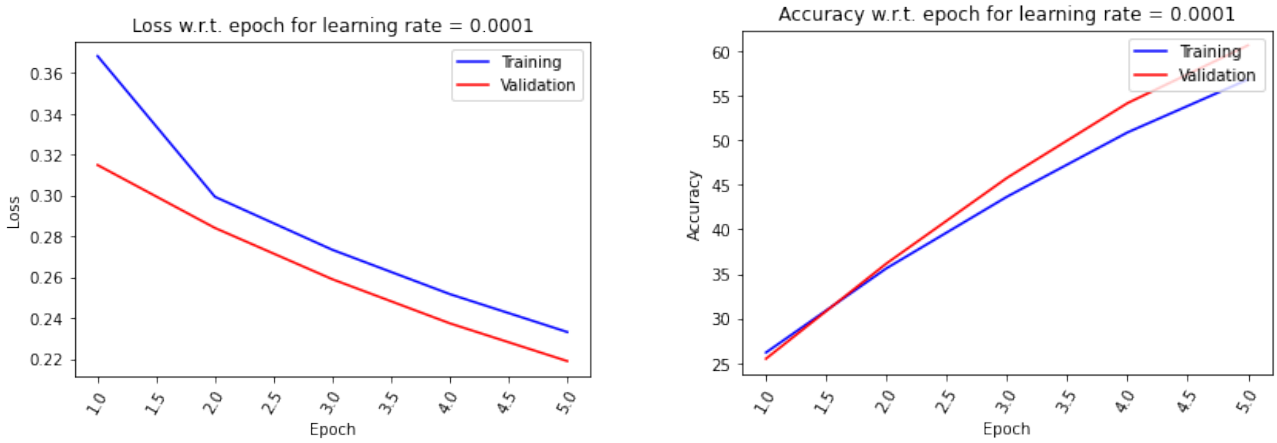
(c) Model Training

Figure 4: Code snippet for MNIST dataset

Question 5 Further, the mnist dataset was trained using stochastic gradient descent.

1. For learning rate of 0.0001, cross entropy loss and accuracy is observed over 5 epochs for training and validation set. Figure 5 depicts the visualization. Accuracy is almost similar until epoch 2, there after the accuracy diverges and increases for validation set. In the loss curve, there is a significant gap between the training and validation set, by which it can conclude that there is a possibility of under fitting. With this result,

it can also be concluded that the model was capable of further learning and the training process was halted prematurely(more likely scenario).



(a) Cross Entropy Loss (b) Accuracy

Figure 5: Comparison of Training Vs Validation

2. Stochastic gradient descent is run 3 times for 5 epochs for a learning rate of 0.001. Mean and Standard deviation per epoch is plotted for these 3 runs of SGD [2]. From the Figure 6, it can be seen that there is variation in the values of loss obtained at a particular epoch. This variation is due to the random values of weights initialization of the network parameters.

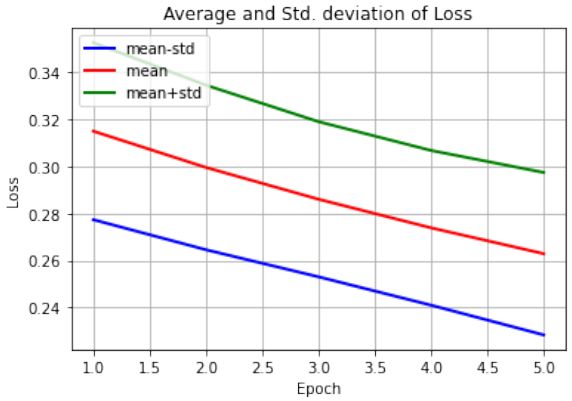
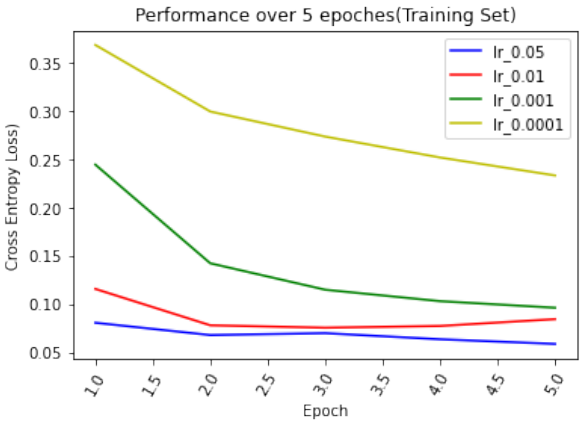


Figure 6: Mean and Std Deviation for SGD(3 times)

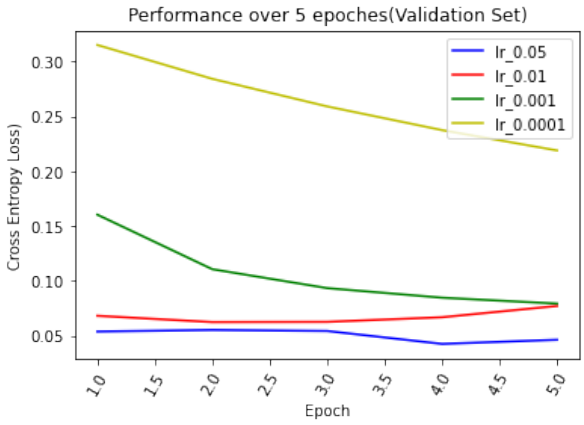
3. Learning rate comparison was done for 4 different learning rates i.e. 0.05, 0.01, 0.001, 0.0001 on the training data. Figure 7 shows the results. It is seen that a lower learning rate performs better both on the training and validation set. However, further lowering the learning can also be investigated as it may or may not give good performance. Here, the learning rate of 0.001 shows that the loss starts with the higher value and then reduces by a significant value. While loss for the rate 0.0001 starts with a much higher value of loss and then reduces by a significant value. These two rates are further tested during training the network.

4. To finally check the hyper-parameters, the positional parameter 'final' is updated to True in order to load the full training and canonical dataset. There are 60,000 and 10,000 records respectively in training and canonical set. Epoch = 20 and Learning Rate = 0.0001 is chosen to conduct this test. Figure 8 shows the results.

[2]Stochastic Gradient Descent

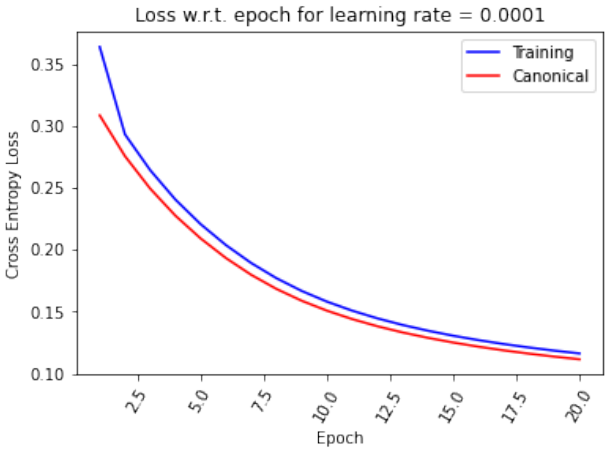


(a) Learning Rates Comparison(Training Set)

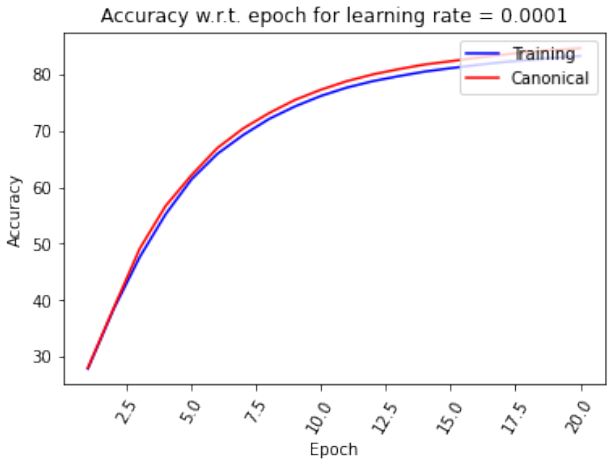


(b) Learning Rates Comparison(Validation Set

Figure 7: Comparison of Learning Rates



(a) Loss Curve for Training and Canonical Set



(b) Accuracy for Training and Canonical Set

Figure 8: Performance on Full Training and Canonical Set

Here, it can seen that the model performs better when running over 20 epochs. For loss curve, the gap between the training canonical set is reduced significantly. Improvements over loss curve can further be done by various means and further tuning the hyper parameters. First,changing the random seed values for weights initialization. Second, further training the neural network for higher epochs. On the right side, the accuracy performs better than earlier evaluated parameters setting. Canonical set performs slightly better than the training set with the increased number of epochs. For full training data the accuracy at epoch 1 is 27.88% while at epoch 20 is 83.25%. However, for canonical data the accuracy is 28.04% at epoch 1 and at epoch 20 is 84.61%. The accuracy can further be increased for MNIST dataset.

Appendix

Question 2

```
def categorical_cross_entropy(predicted, actual):
    sum_score = 0.0
    for i in range(len(actual)):
        sum_score += actual[i] * log(1e-15 + predicted[i])
    mean_sum_score = 1.0 / len(actual) * sum_score
    return -2 * mean_sum_score

def initialize_parameters():
    """
    Returns:
    params -- python dictionary containing parameters:

    """
    W = ([[1., 1., 1.], [-1., -1., -1.]])
    b = ([0., 0., 0.])
    V = ([[1., 1.], [-1., -1.],[-1., -1.]])
    c = ([0., 0.])
    k1 = ([0., 0., 0.])
    h = ([0., 0., 0.])
    k2 = ([0., 0.])
    y = ([0., 0.])

    parameters = {"W": W,
                  "b": b,
                  "V": V,
                  "c": c,
                  "k1": k1,
                  "h": h,
                  "k2": k2,
                  "y": y}

    return parameters

def forward_propagation(X, parameters):
    """
    Returns:
    y -- The softmax output of the second activation
    cache -- a dictionary containing "k1", "h", "k2" and "y"
    """
    # Retrieve each parameter from the dictionary "parameters"
    W = parameters["W"]
    b = parameters["b"]
    V = parameters["V"]
    c = parameters["c"]
    k1 = parameters["k1"]
    h = parameters["h"]
    k2 = parameters["k2"]
    y = parameters["y"]

    # Implement Forward Propagation to calculate y (output) (probabilities)
    transpose_W = list(map(list, zip(*W)))
    transpose_V = list(map(list, zip(*V)))

    for j in range(len(transpose_W)):
        for i in range(len(X)):
            k1[j] += W[i][j] * X[i]
        k1[j] += b[j]
    print(f"k1 = {k1}")

    for i in range(len(k1)):
        h[i] = sigmoid(k1[i])
    print(f"h = {h}")

    for j in range(len(transpose_V)):
        for i in range(len(h)):
            k2[j] += V[i][j] * h[i]
        k2[j] += c[j]
    print(f"k2 = {k2}")

    y = softmax(k2)
    print(f"y = {y}")

    cache = {"k1": k1,
             "h": h,
             "k2": k2,
             "y": y}

    return y, cache

def backward_propagation(parameters, cache ,X, Y):
    """
    Returns:
    grads -- python dictionary containing gradients with respect to different parameters
    """
    # First, W and V and retrived from the dictionary "parameters".
    W = parameters["W"]
    b = parameters["b"]
    V = parameters["V"]
    c = parameters["c"]

    transpose_W = list(map(list, zip(*W)))
    transpose_V = list(map(list, zip(*V)))
```

```
    # Retrieved also k1 and k2 from dictionary "cache".
    k1 = cache["k1"]
    h = cache["h"]
    k2 = cache["k2"]
    y = cache["y"]

    dW = ([[0., 0., 0.], [0., 0., 0.]])
    dk1 = ([0., 0., 0.])
    db = ([0., 0., 0.])
    dh = ([0., 0., 0.])
    dV = ([[0., 0.], [0., 0.],[0., 0.]])
    dk2 = ([0., 0.])
    dc = ([0., 0.])

    dk2 = y - Y

    for i in range(len(transpose_W)):
        for j in range(len(k2)):
            dV[i][j] = dk2[j] * h[i]
            dh[i] += dk2[j] * V[i][j]

    dc = dk2

    for i in range(len(transpose_W)):
        dk1[i] = dh[i] * h[i] * (1 - h[i])

    for j in range(len(transpose_W)):
        for i in range(len(X)):
            dW[i][j] = dk1[j] * X[i]
        db[j] = dk1[j]


    grads = {"dW": dW,
             "db": db,
             "dV": dV,
             "dc": dc}

    return grads
```

Question 3

```
Neural network architecture methods are same as that for Question2. Some of the addition code is mentioned here.

#Loading Data
(xtrain, ytrain), (xtest, ytest), num_classes = load_synth()
#Normalization
xtrain = (xtrain/255).astype('float32')
xtest = (xtest/255).astype('float32')
# One-Hot Encoding
#Train Set
onehot_y = np.zeros((ytrain.size, ytrain.max()+1))
onehot_y[np.arange(ytrain.size),ytrain] = 1
# Test Set
onehot_y_test = np.zeros((ytest.size, ytest.max()+1))
onehot_y_test[np.arange(ytest.size),ytest] = 1

#Cross entropy
def cross_entropy(p,y):
    log_likelihood = 0.0
    for i in range(len(p)):
        log_likelihood += -1 * y[i] * log(p[i])

    loss = log_likelihood
    return loss
#Stochastic Gradient Descent
def nn_model(X, Y, epochs = 5):

    parameters = initialize_weights()

    loss = list()
    m = len(X)
    cost_all = list()
    # Loop (gradient descent)

    for i in range(epochs):
        cost = 0
        for data in range(len(X)):

            output, cache = forward_propagation_synth(X[data], parameters)

            cost += cross_entropy(output, Y[data])
            cost_all.append(cross_entropy(output, Y[data]))

            grads = backward_propagation_synth(parameters, cache, X[data], Y[data])

            parameters = update_parameters_synth(parameters, grads)

        loss.append(cost/m)
        print (f"Loss␣after␣iteration␣{i}:␣{loss[i]}")

    return loss, cost_all
```

Question 4

```
# Data Load
(xtrain, ytrain), (xtest, ytest), digits = load_mnist()
# Normalization
xtrain = (xtrain/255).astype('float32')
xtest = (xtest/255).astype('float32')

#One Hot Encoding
onehot_y = np.zeros((ytrain.size, ytrain.max()+1))
onehot_y[np.arange(ytrain.size),ytrain] = 1
# Test Set
onehot_y_test = np.zeros((ytest.size, ytest.max()+1))
onehot_y_test[np.arange(ytest.size),ytest] = 1

ytrain = onehot_y
ytest = onehot_y_test

class DeepNeuralNetwork():
    def __init__(self, xtrain, sizes, epochs=5, l_rate=0.001):
        self.sizes = sizes
        self.epochs = epochs
        self.records = xtrain.shape[0]
        self.l_rate = l_rate

        # we save all parameters in the neural network in this dictionary
        self.params = self.initialization()

    def sigmoid(self, x, derivative=False):
        if derivative:
            return (np.exp(-x))/((np.exp(-x)+1)**2)
        return 1/(1 + np.exp(-x))

    def softmax(self, x, derivative=False):
        # Numerically stable with large exponentials
        exps = np.exp(x - x.max())
        if derivative:
            return exps / np.sum(exps, axis=0) * (1 - exps / np.sum(exps, axis=0))
        return exps / np.sum(exps, axis=0)

    def initialization(self):
        # number of nodes in each layer
        input_layer=self.sizes[0]
        hidden_1=self.sizes[1]
        output_layer=self.sizes[2]
        np.random.seed(2)

        params = {
            'W1':np.random.randn(input_layer,hidden_1) * np.sqrt(1. / hidden_1),
            'W2':np.random.randn(hidden_1,output_layer) * np.sqrt(1. / output_layer),
            'b1': np.zeros((1,300)),
            'b2': np.zeros((1,10))
        }

        return params

    def forward_pass(self, x_train):
        params = self.params

        W1 = params["W1"]
        b1 = params["b1"]
        W2 = params["W2"]
        b2 = params["b2"]


        # input layer activations becomes sample
        params['A0'] = x_train
        # input layer to hidden layer 1
        params['Z1'] = np.dot(W1.T,params['A0']) + params['b1']
        params['A1'] = self.sigmoid(params['Z1'])
        # hidden layer 1 to output layer
        params['Z2'] = np.dot(params['A1'],params["W2"]) + params['b2']
        params['A2'] = self.softmax(params['Z2'])

        return params['A2']

    def backward_pass(self, y_train, output):

        params = self.params
        change_w = {}

        #print(params.keys())
        A2 = params['A2']
        A1 = params['A1']
        A0 = params['A0']
        Z2 = params['Z2']
        Z1 = params['Z1']
        W2 = params['W2']
        W1 = params['W1']
        b2 = params['b2']
        b1 = params['b1']


        dZ2 = A2-y_train
        dW2 = dZ2 * A1.T
        db2 = np.sum(dZ2, axis = 1, keepdims = True)
        dZ1 = np.dot(dZ2, W2.T) * (A1*(1-A1))
        dW1 = dZ1.T * A0.T
        dW1 = dW1.T
        db1 = np.sum(dZ1, axis = 1, keepdims = True)
```

```
        change_w['dZ2'] = dZ2
        change_w['dW2'] = dW2
        change_w['db2'] = db2
        change_w['dZ1'] = dZ1
        change_w['dW1'] = dW1
        change_w['db1'] = db1

        return change_w

    def update_network_parameters(self, changes_to_w):

        W1 = self.params["W1"]
        b1 = self.params["b1"]
        W2 = self.params["W2"]
        b2 = self.params["b2"]

        dW1 = changes_to_w["dW1"]
        db1 = changes_to_w["db1"]
        dW2 = changes_to_w["dW2"]
        db2 = changes_to_w["db2"]

        W1 = W1 - dW1 * self.l_rate
        b1 = b1 - db1 * self.l_rate
        W2 = W2 - dW2 * self.l_rate
        b2 = b2 - db2 * self.l_rate

        self.params['W1'] = W1
        self.params['W2'] = W2
        self.params['b1'] = b1
        self.params['b2'] = b2


    def compute_cost(self, A2, Y):

        m = Y.shape[0] # number of example

        cost = -1/ m*np.sum(np.multiply(np.log(A2),Y) + np.multiply(np.log(1-A2), (1-Y)))

        cost = np.squeeze(cost)
        assert(isinstance(cost, float))

        return cost


    def compute_accuracy(self, x_val, y_val):

        predictions = []

        for x, y in zip(x_val, y_val):
            output = self.forward_pass(x)
            pred = np.argmax(output)
            predictions.append(pred == np.argmax(y))

        return np.mean(predictions)


    def train(self, x_train, y_train, x_val, y_val):

        train_loss = list()
        validation_loss = list()
        train_accuracy = list()
        validation_accurracy =  list()
        start_time = time.time()
        m = x_train.shape[0]
        n = x_val.shape[0]

        for iteration in range(self.epochs):
            loss_train = 0
            for x,y in zip(x_train, y_train):
                output = self.forward_pass(x)
                changes_to_w = self.backward_pass(y, output)
                self.update_network_parameters(changes_to_w)
                loss_train = loss_train + self.compute_cost(output,y)
            cost_train = loss_train/m
            train_loss.append(cost_train)

            loss_val = 0
            for x,y in zip(x_val, y_val):
                output = self.forward_pass(x)
                changes_to_w = self.backward_pass(y, output)
                self.update_network_parameters(changes_to_w)
                loss_val = loss_val + self.compute_cost(output,y)
            cost_val = loss_val/n
            validation_loss.append(cost_val)

            accuracy = self.compute_accuracy(x_train, y_train)
            train_accuracy.append(accuracy*100)
            print('Epoch: {0}, Time Spent: {1:.2f}s, Accuracy: {2:.2f}%'.format(
                iteration+1, time.time() - start_time, accuracy * 100))

            accuracy = self.compute_accuracy(x_val, y_val)
            validation_accurracy.append(accuracy*100)
            print('Epoch: {0}, Time Spent: {1:.2f}s, Accuracy: {2:.2f}%'.format(
                iteration+1, time.time() - start_time, accuracy * 100))

        return train_loss, validation_loss, train_accuracy, validation_accurracy

dnn = DeepNeuralNetwork(xtrain,sizes=[784, 300, 10], l_rate=0.05)
train_loss, valid_loss, train_accuracy, valid_accuracy =  dnn.train(xtrain, ytrain, xtest, ytest)
```