Lab 5 D7041E: Artificial Neural Network and Backpropagation

Group 30: Sergio Serrano Hernández (seser-1) and Nicolas Schröder (nicsch-4)

In [1]:
```
import numpy as np
```

In [2]:
```
#Calculate all non-linear activations
def f_sigmoid(X, deriv=False):
    if not deriv:
        return 1 / (1 + np.exp(-X))
    else:
        return f_sigmoid(X)*(1 - f_sigmoid(X))


def f_softmax(X):
    Z = np.sum(np.exp(X), axis=1)
    Z = Z.reshape(Z.shape[0], 1)
    return np.exp(X) / Z
```

In [3]:
```
def exit_with_err(err_str):
    print(err_str)
    sys.exit(1)
```

In [4]:
```
#Functionality of a single hidden layer
class Layer:
    def __init__(self, size, batch_size, is_input=False, is_output=False,
                 activation=f_sigmoid):
        self.is_input = is_input
        self.is_output = is_output

        # Z is the matrix that holds output values
        self.Z = np.zeros((batch_size, size[0]))
        # The activation function is an externally defined function (with a
        # derivative) that is stored here
        self.activation = activation

        # W is the outgoing weight matrix for this layer
        self.W = None
        # S is the matrix that holds the inputs to this layer
        self.S = None
        # D is the matrix that holds the deltas for this layer
        self.D = None
        # Fp is the matrix that holds the derivatives of the activation function
        self.Fp = None

        if not is_input:
            self.S = np.zeros((batch_size, size[0]))
            self.D = np.zeros((batch_size, size[0]))

        if not is_output:
            self.W = np.random.normal(size=size, scale=1E-4)

        if not is_input and not is_output:
            self.Fp = np.zeros((size[0], batch_size))

    def forward_propagate(self):
        if self.is_input:
            return self.Z.dot(self.W)

        self.Z = self.activation(self.S)
        if self.is_output:
            return self.Z
        else:
            # For hidden layers, we add the bias values here
            self.Z = np.append(self.Z, np.ones((self.Z.shape[0], 1)), axis=1)
            self.Fp = self.activation(self.S, deriv=True).T
            return self.Z.dot(self.W)
```

In [5]:
```
class MultiLayerPerceptron:
    def __init__(self, layer_config, batch_size=100):
        self.layers = []
        self.num_layers = len(layer_config)
        self.minibatch_size = batch_size

        for i in range(self.num_layers-1):
            if i == 0:
                print ("Initializing input layer with size {0}.".format(layer_config[i]))
                # Here, we add an additional unit at the input for the bias
                # weight.
                self.layers.append(Layer([layer_config[i]+1, layer_config[i+1]],
                                         batch_size,
                                         is_input=True))
            else:
                print ("Initializing hidden layer with size {0}.".format(layer_config[i]))
                # Here we add an additional unit in the hidden layers for the
                # bias weight.
                self.layers.append(Layer([layer_config[i]+1, layer_config[i+1]],
                                         batch_size,
                                         activation=f_sigmoid))

        print ("Initializing output layer with size {0}.".format(layer_config[-1]))
        self.layers.append(Layer([layer_config[-1], None],
                                 batch_size,
                                 is_output=True,
                                 activation=f_softmax))
        print ("Done!")

    def forward_propagate(self, data):
        # We need to be sure to add bias values to the input
        self.layers[0].Z = np.append(data, np.ones((data.shape[0], 1)), axis=1)

        for i in range(self.num_layers-1):
            self.layers[i+1].S = self.layers[i].forward_propagate()
        return self.layers[-1].forward_propagate()

    def backpropagate(self, yhat, labels):

        #exit_with_err("FIND ME IN THE CODE, What is computed in the next line of code?\n")

        self.layers[-1].D = (yhat - labels).T
        for i in range(self.num_layers-2, 0, -1):
            # We do not calculate deltas for the bias values
            W_nobias = self.layers[i].W[0:-1, :]

            #exit_with_err("FIND ME IN THE CODE, What does this 'for' loop do?\n")


            self.layers[i].D = W_nobias.dot(self.layers[i+1].D) * self.layers[i].Fp

    def update_weights(self, eta):
        for i in range(0, self.num_layers-1):
            W_grad = -eta*(self.layers[i+1].D.dot(self.layers[i].Z)).T
            self.layers[i].W += W_grad

    def evaluate(self, train_data, train_labels, test_data, test_labels,
                 num_epochs=70, eta=0.05, eval_train=False, eval_test=True):

        N_train = len(train_labels)*len(train_labels[0])
        N_test = len(test_labels)*len(test_labels[0])

        print ("Training for {0} epochs...".format(num_epochs))
        for t in range(0, num_epochs):
            out_str = "[{0:4d}] ".format(t)

            for b_data, b_labels in zip(train_data, train_labels):
                output = self.forward_propagate(b_data)
                self.backpropagate(output, b_labels)

                #exit_with_err("FIND ME IN THE CODE, How does weight update is implemented? What is eta?\n")

                self.update_weights(eta=eta)

            if eval_train:
                errs = 0
                for b_data, b_labels in zip(train_data, train_labels):
                    output = self.forward_propagate(b_data)
                    yhat = np.argmax(output, axis=1)
                    errs += np.sum(1-b_labels[np.arange(len(b_labels)), yhat])

                out_str = ("{0} Training error: {1:.5f}".format(out_str,
                                                           float(errs)/N_train))

            if eval_test:
                errs = 0
                for b_data, b_labels in zip(test_data, test_labels):
                    output = self.forward_propagate(b_data)
                    yhat = np.argmax(output, axis=1)
                    errs += np.sum(1-b_labels[np.arange(len(b_labels)), yhat])
                out_str = ("{0} Test error: {1:.5f}").format(out_str,
                                                       float(errs)/N_test)

            print (out_str)
```

Task 1.

Answers to the questions of the code:

1. The line `self.layers[-1].D = (yhat - labels).T` calculates the error in the last layer which will be backpropagated to calculate all the Deltas in the previous steps, yhat is the prediction made by the network and the 'labels' are the correct instances.

2. `self.layers[i].D = W_nobias.dot(self.layers[i+1].D) * self.layers[i].Fp` — The purpose of this loop is to calculate the deltas for each hidden layer, excluding the input layer. These deltas are then used to update the weights during the backward pass (the backpropagation algorithm).

3. How is the weight update implemented? What is eta? The weight update is implemented through a function "update_weights" that calculates the gradient of each layer's weights and updates them. eta is the learning rate, it controls how big the steps of the weight updates are.

In [6]:
```
def label_to_bit_vector(labels, nbits):
    bit_vector = np.zeros((labels.shape[0], nbits))
    for i in range(labels.shape[0]):
        bit_vector[i, labels[i]] = 1.0

    return bit_vector
```

In [7]:
```
def create_batches(data, labels, batch_size, create_bit_vector=False):
    N = data.shape[0]
    print ("Batch size {0}, the number of examples {1}.".format(batch_size,N))

    if N % batch_size != 0:
        print ("Warning in create_minibatches(): Batch size {0} does not " \
               "evenly divide the number of examples {1}.".format(batch_size,N))
    chunked_data = []
    chunked_labels = []
    idx = 0
    while idx + batch_size <= N:
        chunked_data.append(data[idx:idx+batch_size, :])
        if not create_bit_vector:
            chunked_labels.append(labels[idx:idx+batch_size])
        else:
            bit_vector = label_to_bit_vector(labels[idx:idx+batch_size], 10)
            chunked_labels.append(bit_vector)

        idx += batch_size

    return chunked_data, chunked_labels
```

In [8]:
```
def prepare_for_backprop(batch_size, Train_images, Train_labels, Va

    print ("Creating data...")
    batched_train_data, batched_train_labels = create_batches(Train
                                                              batch_size,
                                                              create_bit_vector=Tru
    batched_valid_data, batched_valid_labels = create_batches(Valid
                                                              batch_size,
                                                              create_bit_vector=Tru
    print ("Done!")


    return batched_train_data, batched_train_labels,  batched_valid
```

In [9]:
```
from keras.datasets import mnist
```

In [10]:
```
(Xtr, Ltr), (X_test, L_test)=mnist.load_data()

Xtr = Xtr.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
Xtr = Xtr.astype('float32')
X_test = X_test.astype('float32')
Xtr /= 255
X_test /= 255
print(Xtr.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

```
60000 train samples
10000 test samples
```

In [11]:
```
batch_size=100;

train_data, train_labels, valid_data, valid_labels=prepare_for_back

mlp = MultiLayerPerceptron(layer_config=[784, 100, 100, 10], batch_

mlp.evaluate(train_data, train_labels, valid_data, valid_labels,
             eval_train=True)

print("Done:)\n")
```

```
Creating data...
Batch size 100, the number of examples 60000.
Batch size 100, the number of examples 10000.
Done!
Initializing input layer with size 784.
Initializing hidden layer with size 100.
Initializing hidden layer with size 100.
Initializing output layer with size 10.
Done!
Training for 70 epochs...
[   0]  Training error: 0.73552 Test error: 0.72000
[   1]  Training error: 0.08963 Test error: 0.08910
[   2]  Training error: 0.05932 Test error: 0.06110
[   3]  Training error: 0.04727 Test error: 0.05160
[   4]  Training error: 0.04153 Test error: 0.04770
[   5]  Training error: 0.03540 Test error: 0.04290
[   6]  Training error: 0.03230 Test error: 0.04060
[   7]  Training error: 0.02750 Test error: 0.03740
[   8]  Training error: 0.02318 Test error: 0.03580
[   9]  Training error: 0.02068 Test error: 0.03350
[  10]  Training error: 0.01968 Test error: 0.03520
[  11]  Training error: 0.01933 Test error: 0.03350
[  12]  Training error: 0.01695 Test error: 0.03250
[  13]  Training error: 0.01668 Test error: 0.03210
[  14]  Training error: 0.02027 Test error: 0.03550
[  15]  Training error: 0.01318 Test error: 0.03380
[  16]  Training error: 0.01235 Test error: 0.03150
[  17]  Training error: 0.01405 Test error: 0.03320
[  18]  Training error: 0.01513 Test error: 0.03210
[  19]  Training error: 0.01195 Test error: 0.03170
[  20]  Training error: 0.01092 Test error: 0.03130
[  21]  Training error: 0.01153 Test error: 0.03180
[  22]  Training error: 0.00987 Test error: 0.03190
[  23]  Training error: 0.00915 Test error: 0.03200
[  24]  Training error: 0.00973 Test error: 0.03120
[  25]  Training error: 0.00877 Test error: 0.02800
[  26]  Training error: 0.00710 Test error: 0.02920
[  27]  Training error: 0.00738 Test error: 0.03090
[  28]  Training error: 0.00415 Test error: 0.02840
[  29]  Training error: 0.00842 Test error: 0.03180
[  30]  Training error: 0.00410 Test error: 0.02930
[  31]  Training error: 0.00350 Test error: 0.02910
[  32]  Training error: 0.00252 Test error: 0.02870
[  33]  Training error: 0.01137 Test error: 0.03320
[  34]  Training error: 0.00740 Test error: 0.03040
[  35]  Training error: 0.00572 Test error: 0.02980
[  36]  Training error: 0.00512 Test error: 0.02920
[  37]  Training error: 0.00483 Test error: 0.03020
[  38]  Training error: 0.00608 Test error: 0.02940
[  39]  Training error: 0.00370 Test error: 0.02980
[  40]  Training error: 0.00345 Test error: 0.02950
[  41]  Training error: 0.00265 Test error: 0.02730
[  42]  Training error: 0.00157 Test error: 0.02630
[  43]  Training error: 0.00158 Test error: 0.02680
[  44]  Training error: 0.00585 Test error: 0.02970
[  45]  Training error: 0.00408 Test error: 0.02690
[  46]  Training error: 0.00338 Test error: 0.02800
[  47]  Training error: 0.00187 Test error: 0.02730
[  48]  Training error: 0.00300 Test error: 0.02750
[  49]  Training error: 0.00073 Test error: 0.02590
[  50]  Training error: 0.00082 Test error: 0.02640
[  51]  Training error: 0.00023 Test error: 0.02490
[  52]  Training error: 0.00008 Test error: 0.02520
[  53]  Training error: 0.00003 Test error: 0.02510
[  54]  Training error: 0.00003 Test error: 0.02510
[  55]  Training error: 0.00003 Test error: 0.02460
[  56]  Training error: 0.00003 Test error: 0.02480
[  57]  Training error: 0.00002 Test error: 0.02520
[  58]  Training error: 0.00002 Test error: 0.02530
[  59]  Training error: 0.00002 Test error: 0.02530
[  60]  Training error: 0.00002 Test error: 0.02510
[  61]  Training error: 0.00002 Test error: 0.02510
[  62]  Training error: 0.00002 Test error: 0.02490
[  63]  Training error: 0.00002 Test error: 0.02490
[  64]  Training error: 0.00002 Test error: 0.02490
[  65]  Training error: 0.00002 Test error: 0.02500
[  66]  Training error: 0.00002 Test error: 0.02510
[  67]  Training error: 0.00002 Test error: 0.02530
[  68]  Training error: 0.00000 Test error: 0.02510
[  69]  Training error: 0.00002 Test error: 0.02520
Done:)
```

Task 2: Run the code with the suggested configuration of the hyperparameters: number of epochs = 70 and learning rate = 0.05. What is the classification accuracy?

In [14]:
```
output = mlp.forward_propagate(X_test)
y_hat = np.argmax(output, axis = 1)
accuracy = np.sum(y_hat == L_test)/len(L_test)
print(f"Accuracy: {accuracy:.4f}")
```

```
Accuracy: 0.9748
```

The classification accuracy when training the MLP for 70 epochs witha a learning rate of 0.05 is 97.48%

Task 3: Run the code with Learning rate =0.005 and Learning rate =0.5. Explain the observed differences in the functionality of the multi-layer perceptron.

In [18]:
```
mlp = MultiLayerPerceptron(layer_config=[784, 100, 100, 10], batch_size=batch_size)

learning_rates = [0.005, 0.5]

for i in range(2):
    mlp.evaluate(train_data, train_labels, valid_data, valid_labels, num_epochs=70, eta=learning_rates[i], eval_train=F
    output = mlp.forward_propagate(X_test)
    y_hat = np.argmax(output, axis = 1)
    accuracy = np.sum(y_hat == L_test)/len(L_test)
    print(f"The Accuracy when training with a learning rate of {learning_rates[i]}  is: {accuracy:.4f}")
```

```
Initializing input layer with size 784.
Initializing hidden layer with size 100.
Initializing hidden layer with size 100.
Initializing output layer with size 10.
Done!
The Accuracy when training with a learning rate of 0.005  is: 0.9749
C:\Users\sexysehe\AppData\Local\Temp\ipykernel_22028\3670988153.py:4: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-X))
The Accuracy when training with a learning rate of 0.5  is: 0.0974
```

As we can see by the accuracies gotten for the different learning rates, increasing it to 0.5 has made the accuracy be much lower when testing the MLP. This could be because since the learning rate is so large it makes the optimization process jump around the minima.

Task 4

Extend the code implementing the ReLU output function. Run the perceptron with the suggested by default configuration of hyperparameters: number of epochs = 70 and learning rate =0.05. What is the classification accuracy?

We will first change the parts in the object oriented programming that are affected by this, you can see how below we change the activations from sigmoids to the relu that we have defined.

In [44]:
```
def f_relu(X, deriv=False):
    if not deriv:
        return np.maximum(0, X)
    else:
        return np.where(X > 0, 1, 0)
```

In [21]:
```
#Functionality of a single hidden layer
class Layer:
    def __init__(self, size, batch_size, is_input=False, is_output=False,
                 activation=f_relu):
        self.is_input = is_input
        self.is_output = is_output

        # Z is the matrix that holds output values
        self.Z = np.zeros((batch_size, size[0]))
        # The activation function is an externally defined function (with a
        # derivative) that is stored here
        self.activation = activation

        # W is the outgoing weight matrix for this layer
        self.W = None
        # S is the matrix that holds the inputs to this layer
        self.S = None
        # D is the matrix that holds the deltas for this layer
        self.D = None
        # Fp is the matrix that holds the derivatives of the activation function
        self.Fp = None

        if not is_input:
            self.S = np.zeros((batch_size, size[0]))
            self.D = np.zeros((batch_size, size[0]))

        if not is_output:
            self.W = np.random.normal(size=size, scale=1E-4)

        if not is_input and not is_output:
            self.Fp = np.zeros((size[0], batch_size))

    def forward_propagate(self):
        if self.is_input:
            return self.Z.dot(self.W)

        self.Z = self.activation(self.S)
        if self.is_output:
            return self.Z
        else:
            # For hidden layers, we add the bias values here
            self.Z = np.append(self.Z, np.ones((self.Z.shape[0], 1)), axis=1)
            self.Fp = self.activation(self.S, deriv=True).T
            return self.Z.dot(self.W)
```

In [34]:
```
class MultiLayerPerceptron:
    def __init__(self, layer_config, batch_size=100):
        self.layers = []
        self.num_layers = len(layer_config)
        self.minibatch_size = batch_size

        for i in range(self.num_layers-1):
            if i == 0:
                print ("Initializing input layer with size {0}.".format(layer_config[i]))
                # Here, we add an additional unit at the input for the bias
                # weight.
                self.layers.append(Layer([layer_config[i]+1, layer_config[i+1]],
                                         batch_size,
                                         is_input=True))
            else:
                print ("Initializing hidden layer with size {0}.".format(layer_config[i]))
                # Here we add an additional unit in the hidden layers for the
                # bias weight.
                self.layers.append(Layer([layer_config[i]+1, layer_config[i+1]],
                                         batch_size,
                                         activation=f_relu))

        print ("Initializing output layer with size {0}.".format(layer_config[-1]))
        self.layers.append(Layer([layer_config[-1], None],
                                 batch_size,
                                 is_output=True,
                                 activation=f_softmax))
        print ("Done!")

    def forward_propagate(self, data):
        # We need to be sure to add bias values to the input
        self.layers[0].Z = np.append(data, np.ones((data.shape[0], 1)), axis=1)

        for i in range(self.num_layers-1):
            self.layers[i+1].S = self.layers[i].forward_propagate()
        return self.layers[-1].forward_propagate()

    def backpropagate(self, yhat, labels):

        #exit_with_err("FIND ME IN THE CODE, What is computed in the next line of code?\n")

        self.layers[-1].D = (yhat - labels).T
        for i in range(self.num_layers-2, 0, -1):
            # We do not calculate deltas for the bias values
            W_nobias = self.layers[i].W[0:-1, :]

            #exit_with_err("FIND ME IN THE CODE, What does this 'for' loop do?\n")


            self.layers[i].D = W_nobias.dot(self.layers[i+1].D) * self.layers[i].Fp

    def update_weights(self, eta):
        for i in range(0, self.num_layers-1):
            W_grad = -eta*(self.layers[i+1].D.dot(self.layers[i].Z)).T
            self.layers[i].W += W_grad

    def evaluate(self, train_data, train_labels, test_data, test_labels,
                 num_epochs=70, eta=0.05, eval_train=False, eval_test=True):

        N_train = len(train_labels)*len(train_labels[0])
        N_test = len(test_labels)*len(test_labels[0])

        #print ("Training for {0} epochs...".format(num_epochs))
        for t in range(0, num_epochs):
            out_str = "[{0:4d}] ".format(t)

            for b_data, b_labels in zip(train_data, train_labels):
                output = self.forward_propagate(b_data)
                self.backpropagate(output, b_labels)

                #exit_with_err("FIND ME IN THE CODE, How does weight update is implemented? What is eta?\n")

                self.update_weights(eta=eta)

            if eval_train:
                errs = 0
                for b_data, b_labels in zip(train_data, train_labels):
                    output = self.forward_propagate(b_data)
                    yhat = np.argmax(output, axis=1)
                    errs += np.sum(1-b_labels[np.arange(len(b_labels)), yhat])

                out_str = ("{0} Training error: {1:.5f}".format(out_str,
                                                           float(errs)/N_train))

            if eval_test:
                errs = 0
                for b_data, b_labels in zip(test_data, test_labels):
                    output = self.forward_propagate(b_data)
                    yhat = np.argmax(output, axis=1)
                    errs += np.sum(1-b_labels[np.arange(len(b_labels)), yhat])
                out_str = ("{0} Test error: {1:.5f}").format(out_str,
                                                       float(errs)/N_test)

            print (out_str)
```

Below we will train the MLP using relu activation function, by not initializing the learning rate it stays at the default value of 0.05 which is the one required by this task, and the same goes with the number of epochs which by default is 70.

In [45]:
```
mlp_relu = MultiLayerPerceptron(layer_config=[784, 100, 100, 10], batch_size=batch_size)

mlp_relu.evaluate(train_data, train_labels, valid_data, valid_labels,
             eval_train=True)

print("Done:)\n")
```

```
Initializing input layer with size 784.
Initializing hidden layer with size 100.
Initializing hidden layer with size 100.
Initializing output layer with size 10.
Done!
[    0]  Training error: 0.90137 Test error: 0.90420
[    1]  Training error: 0.90137 Test error: 0.90420
[    2]  Training error: 0.90137 Test error: 0.90420
[    3]  Training error: 0.90137 Test error: 0.90420
[    4]  Training error: 0.90137 Test error: 0.90420
[    5]  Training error: 0.90137 Test error: 0.90420
[    6]  Training error: 0.90137 Test error: 0.90420
[    7]  Training error: 0.90137 Test error: 0.90420
[    8]  Training error: 0.90137 Test error: 0.90420
[    9]  Training error: 0.90137 Test error: 0.90420
[   10]  Training error: 0.90137 Test error: 0.90420
[   11]  Training error: 0.90137 Test error: 0.90420
[   12]  Training error: 0.90137 Test error: 0.90420
[   13]  Training error: 0.90137 Test error: 0.90420
[   14]  Training error: 0.90137 Test error: 0.90420
[   15]  Training error: 0.90137 Test error: 0.90420
[   16]  Training error: 0.90137 Test error: 0.90420
[   17]  Training error: 0.90137 Test error: 0.90420
[   18]  Training error: 0.90137 Test error: 0.90420
[   19]  Training error: 0.90137 Test error: 0.90420
[   20]  Training error: 0.90137 Test error: 0.90420
[   21]  Training error: 0.90137 Test error: 0.90420
[   22]  Training error: 0.90137 Test error: 0.90420
[   23]  Training error: 0.90137 Test error: 0.90420
[   24]  Training error: 0.90137 Test error: 0.90420
[   25]  Training error: 0.90137 Test error: 0.90420
[   26]  Training error: 0.90137 Test error: 0.90420
[   27]  Training error: 0.90137 Test error: 0.90420
[   28]  Training error: 0.90137 Test error: 0.90420
[   29]  Training error: 0.90137 Test error: 0.90420
[   30]  Training error: 0.90137 Test error: 0.90420
[   31]  Training error: 0.90137 Test error: 0.90420
[   32]  Training error: 0.90137 Test error: 0.90420
[   33]  Training error: 0.90137 Test error: 0.90420
[   34]  Training error: 0.90137 Test error: 0.90420
[   35]  Training error: 0.90137 Test error: 0.90420
[   36]  Training error: 0.90137 Test error: 0.90420
[   37]  Training error: 0.90137 Test error: 0.90420
[   38]  Training error: 0.90137 Test error: 0.90420
[   39]  Training error: 0.90137 Test error: 0.90420
[   40]  Training error: 0.90137 Test error: 0.90420
[   41]  Training error: 0.90137 Test error: 0.90420
[   42]  Training error: 0.90137 Test error: 0.90420
[   43]  Training error: 0.90137 Test error: 0.90420
[   44]  Training error: 0.90137 Test error: 0.90420
[   45]  Training error: 0.90137 Test error: 0.90420
[   46]  Training error: 0.90137 Test error: 0.90420
[   47]  Training error: 0.90137 Test error: 0.90420
[   48]  Training error: 0.90137 Test error: 0.90420
[   49]  Training error: 0.90137 Test error: 0.90420
[   50]  Training error: 0.90137 Test error: 0.90420
[   51]  Training error: 0.90137 Test error: 0.90420
[   52]  Training error: 0.90137 Test error: 0.90420
[   53]  Training error: 0.90137 Test error: 0.90420
[   54]  Training error: 0.90137 Test error: 0.90420
[   55]  Training error: 0.90137 Test error: 0.90420
[   56]  Training error: 0.90137 Test error: 0.90420
[   57]  Training error: 0.90137 Test error: 0.90420
[   58]  Training error: 0.90137 Test error: 0.90420
[   59]  Training error: 0.90137 Test error: 0.90420
[   60]  Training error: 0.90137 Test error: 0.90420
[   61]  Training error: 0.90137 Test error: 0.90420
[   62]  Training error: 0.90137 Test error: 0.90420
[   63]  Training error: 0.90137 Test error: 0.90420
[   64]  Training error: 0.90137 Test error: 0.90420
[   65]  Training error: 0.90137 Test error: 0.90420
[   66]  Training error: 0.90137 Test error: 0.90420
[   67]  Training error: 0.90137 Test error: 0.90420
[   68]  Training error: 0.90137 Test error: 0.90420
[   69]  Training error: 0.90137 Test error: 0.90420
Done:)
```

In [46]:

```
output = mlp_relu.forward_propagate(X_test)
y_hat = np.argmax(output, axis = 1)
accuracy = np.sum(y_hat == L_test)/len(L_test)
print(f"Accuracy: {accuracy:.4f}")
```

```
Accuracy: 0.0958
```

The accuracy of the mlp using relu is much lower to the one using the sigmoid activation function, giving us a value of 9.58% which is the final required result of this Lab