Assignment1

February 7, 2023

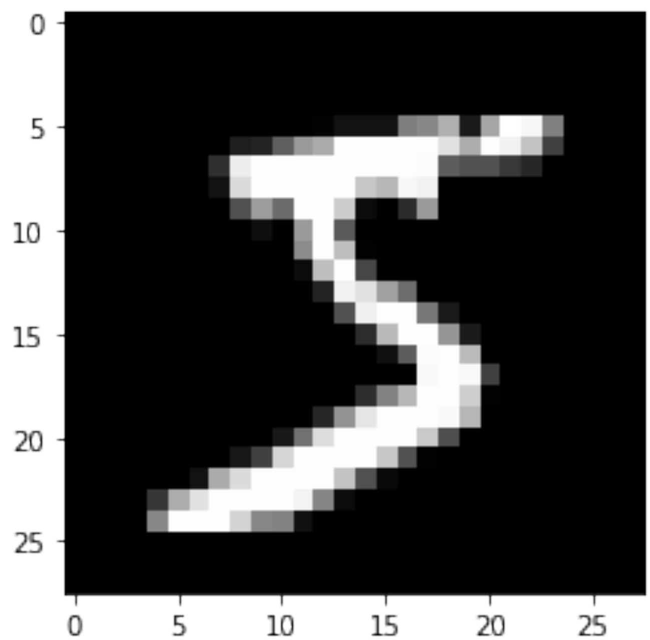
1 Attempt 1

[127]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
from tqdm import tqdm
```

[128]:
```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

[129]:
```
#normalize the data
x_train = x_train/255
x_test = x_test/255
```

[130]:
```
# select an image from the dataset
selected_image = x_train[0]

# view the image
plt.imshow(selected_image, cmap='gray')
plt.show()
```



[131]:
```
#reshape the data so it can fit the neural network model
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

from keras.utils import to_categorical
y_train = to_categorical(y_train, num_classes=10) #reshape into categorical data 0 or 1
y_test = to_categorical(y_test, num_classes=10)
```

[132]:
```
x_train
```

[132]:
```
array([[0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       …,
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.]])
```

[21]:
```
import collections, numpy
train_ans = np.argmax(y_train,axis=1)
collections.Counter(train_ans)
```

[21]:
```
Counter({5: 5421,
         0: 5923,
         4: 5842,
```

```
            1: 6742,
            9: 5949,
            2: 5958,
            3: 6131,
            6: 5918,
            7: 6265,
            8: 5851})
```

[22]:
```
x_train.shape, x_test.shape
```

[22]:
```
((60000, 784), (10000, 784))
```

[233]:
```
def sigmoid(x):
    x = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

def relu(x):
    return(np.maximum(0, x))

def relu_derivative(x):
    x[x<=0] = 0
    x[x>0] = 1
    return x

def leaky_relu(x, alpha=0.01):
    return np.maximum(alpha * x, x)

def leaky_relu_derivative(x, alpha=0.01):
    grad = np.zeros_like(x)
    grad[x >= 0] = 1
    grad[x < 0] = alpha
    return grad


def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1-np.tanh(x)**2

def softmax(x):
    x = x - np.max(x, axis = 1).reshape(x.shape[0],1)
    return np.exp(x) / np.sum(np.exp(x), axis = 1).reshape(x.shape[0],1)
```

```
# def softmax_derivative(x):
#     s = np.exp(x) / np.sum(np.exp(x), axis=0)
#     return np.diagflat(s) - np.dot(s[:, None], s[None, :])
# import numpy as np

def softmax_derivative(x):
    sm = softmax(x)
    sm_diag = np.diagflat(x)
    return np.dot(sm_diag, sm_diag.T) - np.outer(x, x)
# def softmax_derivative(z):
#     s = np.exp(z) / np.sum(np.exp(z), axis=0)
#     grad = np.diag(s) - np.dot(s.reshape(-1, 1), s.reshape(1, -1))
#     return grad
```

[261]:
```
import numpy as np

class NeuralNetwork:
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size,␣
↪learning_rate, batch, activated_func, activated_func_deriv, last_layer_act,␣
↪x_train, y_train, x_test,y_test, num_epochs = 10):
        # initialize weights and biases
        self.weights1 = np.random.randn(input_size, hidden_size1)
        self.weights2 = np.random.randn(hidden_size1, hidden_size2)
        self.weights3 = np.random.randn(hidden_size2, output_size)
        self.bias1 = np.random.randn(hidden_size1)
        self.bias2 = np.random.randn(hidden_size2)
        self.bias3 = np.random.randn(output_size)
        self.learning_rate = learning_rate
        self.batch = batch
        self.activated_func = activated_func
        self.activated_func_deriv = activated_func_deriv
        self.last_layer_act = last_layer_act
        self.x = x_train
        self.y = y_train
        self.inputs = x_train[:self.batch]
        self.outputs = y_train[:self.batch]
        self.x_test = x_test
        self.y_test = y_test
        self.num_epochs = num_epochs
        self.loss = []
        self.acc = []


    def shuffle(self):
        idx = [i for i in range(self.x.shape[0])]
        np.random.shuffle(idx)
        self.x = self.x[idx]
```

```
        self.y = self.y[idx]


    def feedforward(self):
        # feedforward through the first layer
        # print("feedforward weights 1, 2, 3", self.weights1, self.weights2,␣
↪self.weights3)
        self.z1 = np.dot(self.inputs, self.weights1) + self.bias1
        self.layer1 = self.activated_func(self.z1)
        # feedforward through the second layer
        # print("layer shape", self.layer1.shape)
        self.z2 = np.dot(self.layer1, self.weights2) + self.bias2
        self.layer2 = self.activated_func(self.z2)

        self.z3 = np.dot(self.layer2, self.weights3) + self.bias3
        self.output_bar = self.last_layer_act(self.z3)
        # print("weight2 shape", self.weights2.shape, "bias shape", self.bias2.
↪shape)
        before = np.dot(self.layer2, self.weights3) + self.bias3
        # print("before softmax", before[:10])
        # print("output bar", self.output_bar[:10])
        # print("before", before)
        # print("output_bar", self.output_bar)
        return self.output_bar


    def backprop(self):

        # calculate the error for the output layer
        # np.sum(self.outputs * np.log(self.output_bar )
        output_error = -np.sum(self.outputs * np.log(self.output_bar + 1e-10)) /
↪ self.batch
        output_delta = self.output_bar - self.outputs

        # calculate the error for the second hidden layer
        hidden2_error = np.dot(output_delta, self.weights3.T)
        hidden2_delta = hidden2_error * self.activated_func_deriv(self.layer2)

        # calculate the error for the first hidden layer
        hidden1_error = np.dot(hidden2_delta, self.weights2.T)
        hidden1_delta = hidden1_error * self.activated_func_deriv(self.layer1)

        # update the weights and biases
        self.weights3 -= np.dot(self.layer2.T, output_delta) * self.
↪learning_rate
        self.weights2 -= np.dot(self.layer1.T, hidden2_delta) * self.
↪learning_rate
```

```
        self.weights1 -= np.dot(self.inputs.T, hidden1_delta) * self.
↪learning_rate
        self.bias3 -= np.sum(output_delta) * self.learning_rate
        self.bias2 -= np.sum(hidden2_delta) * self.learning_rate
        self.bias1 -= np.sum(hidden1_delta) * self.learning_rate


    def cross_entropy_loss(self):

        loss = -np.sum(self.outputs * np.log(self.output_bar + 1e-10)) / self.
↪batch
        return loss

    def loss_cal(self):
        # self.loss.append(l/(self.input.shape[0]//self.batch))
        return np.mean(np.square(self.outputs - self.output_bar))

    def train(self):
        """
        Train the neural network using the given inputs and outputs
        inputs: array of inputs of shape (number of inputs, number of examples)
        outputs: array of outputs of shape (number of outputs, number of␣
↪examples)
        learning_rate: float, the learning rate to use for the update
        num_epochs: int, the number of times to train the network on the entire␣
↪dataset
        """
        predict = []
        real = []
        # print("start training")
        for epoch in tqdm(range(self.num_epochs)):
            loss = 0
            acc_count = 0
            # self.shuffle()
            # print("epoch 1")
            # print("self.x.shape[0]=", self.x.shape[0])
            # print("self.batch=", self.batch)
            # print("self.x.shape[0]//self.batch-1= ", self.x.shape[0]//self.
↪batch-1)
            for batch in range(self.x.shape[0]//self.batch):
                # print("batch 1")
                start = batch*self.batch
                end = (batch+1)*self.batch
                self.inputs = self.x[start:end]
                self.ouputs = self.y[start:end]
                self.feedforward()
                self.backprop()
```

```
                loss += self.cross_entropy_loss()
                # print("my ouput", self.output_bar[:10])
                predict.append(np.argmax(self.output_bar,axis=1))
                real.append(np.argmax(self.outputs,axis=1))
                # print("correct output", self.outputs[:10])
                # print(np.argmax(self.output_bar,axis=1), np.argmax(self.
↪outputs,axis=1))
                acc_count += np.count_nonzero(np.argmax(self.output_bar,axis=1)␣
↪== np.argmax(self.outputs,axis=1))
            self.loss.append(loss/(self.x.shape[0]))
            self.acc.append(acc_count*100/(self.x.shape[0]))

        return predict, real

    def loss_plot(self):
        plt.figure(dpi = 125)
        plt.plot(self.loss)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")

    def acc_cal(self):
        # print("choose bucket", np.argmax(self.output_bar,axis=1)[:10])
        # print("output", self.outputs[:3])
        # print("right bucket", np.argmax(self.outputs,axis=1)[:10])
        acc_count = np.count_nonzero(np.argmax(self.output_bar,axis=1) == np.
↪argmax(self.outputs,axis=1))
        acc = int(acc_count)/self.outputs.shape[0] * 100
        return acc

    def accuracy_plot(self):
        plt.figure(dpi = 125)
        plt.plot(self.acc)
        plt.xlabel("Epochs")
        plt.ylabel("Accuracy")

    def test(self):
        self.inputs = self.x_test
        self.outputs = self.y_test
        self.feedforward()
        acc = np.count_nonzero(np.argmax(self.output_bar,axis=1) == np.
↪argmax(self.outputs,axis=1)) / self.inputs.shape[0]
        print("Accuracy:", 100 * acc, "%")
        predict = np.argmax(self.output_bar,axis=1)
        real = np.argmax(self.outputs,axis=1)
        return predict, real
```

Code notes: `np.random.randn`: Randomly initialize weights and bias following a normal distribution with mean =0, std = 1

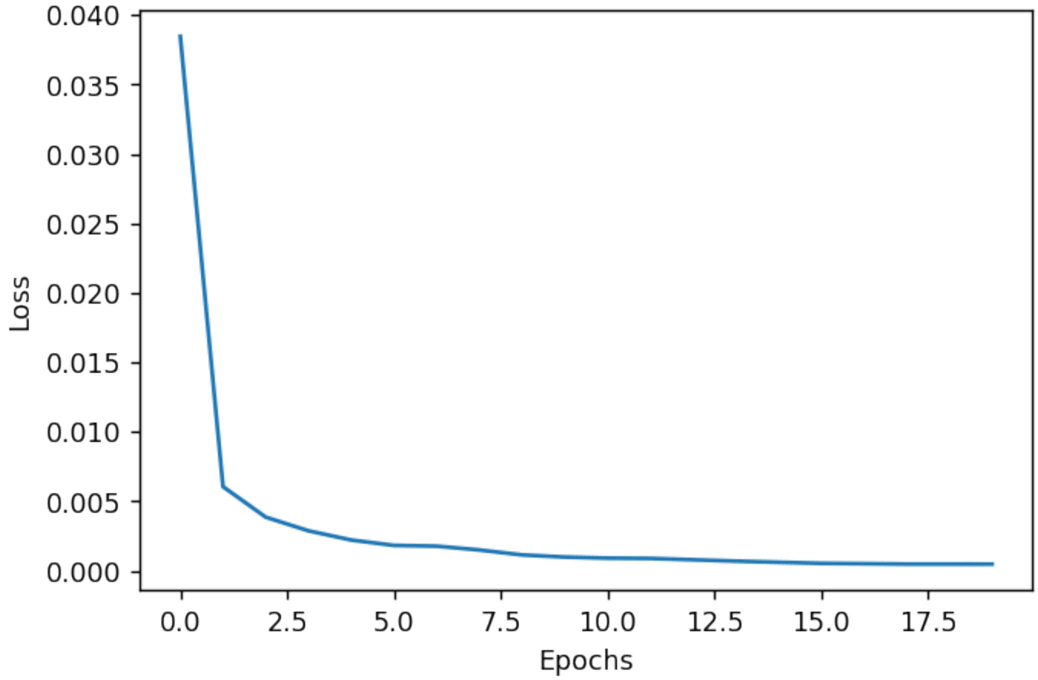
1.1 Test case 1: Select only the samples with label 1

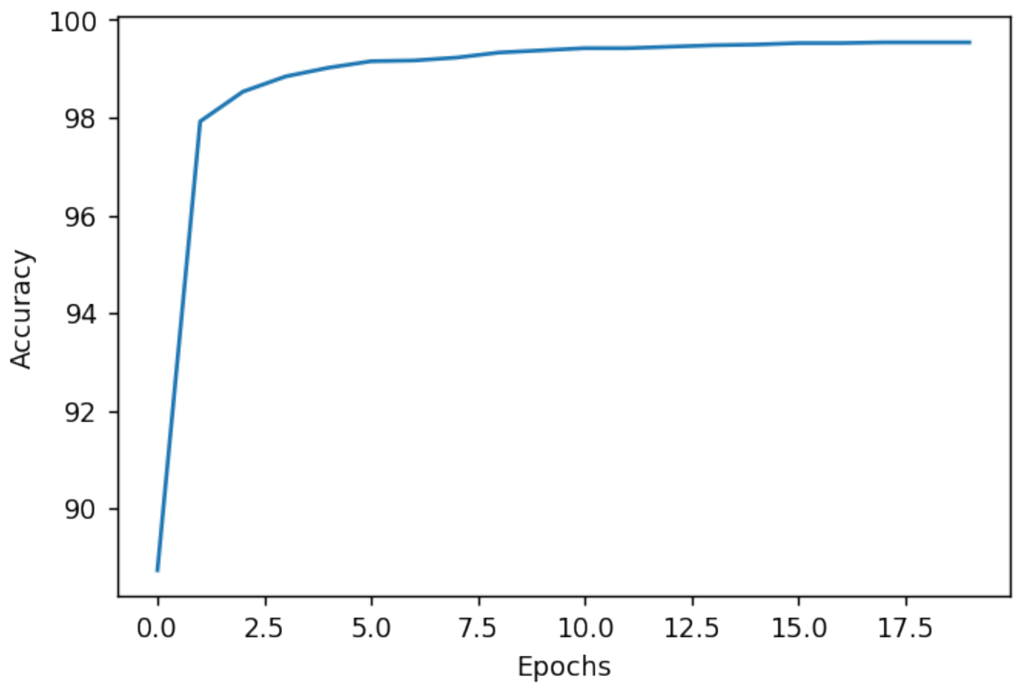
[262]:
```
# #Select only the samples with label 1
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train[y_train == 1]
y_train = y_train[y_train == 1]
x_test = x_test[y_test == 1]
y_test = y_test[y_test == 1]

x_train = x_train/255
x_test = x_test/255
#reshape the data so it can fit the neural network model
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

from keras.utils import to_categorical
y_train = to_categorical(y_train, num_classes=10) #reshape into categorical␣
↪data 0 or 1
y_test = to_categorical(y_test, num_classes=10)

print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

```
(6742, 784) (6742, 10)
(1135, 784) (1135, 10)
```

[263]:
```
nn = NeuralNetwork(784, 256, 128, 10, 0.00001, 64, relu, relu_derivative,␣
↪softmax, x_train, y_train, x_test, y_test, 20)
predict, real = nn.train()
nn.loss_plot()
nn.accuracy_plot()
```

```
100%|          | 20/20 [00:20<00:00,  1.03s/it]
```


0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs


100
98
96
94
92
90
Accuracy
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

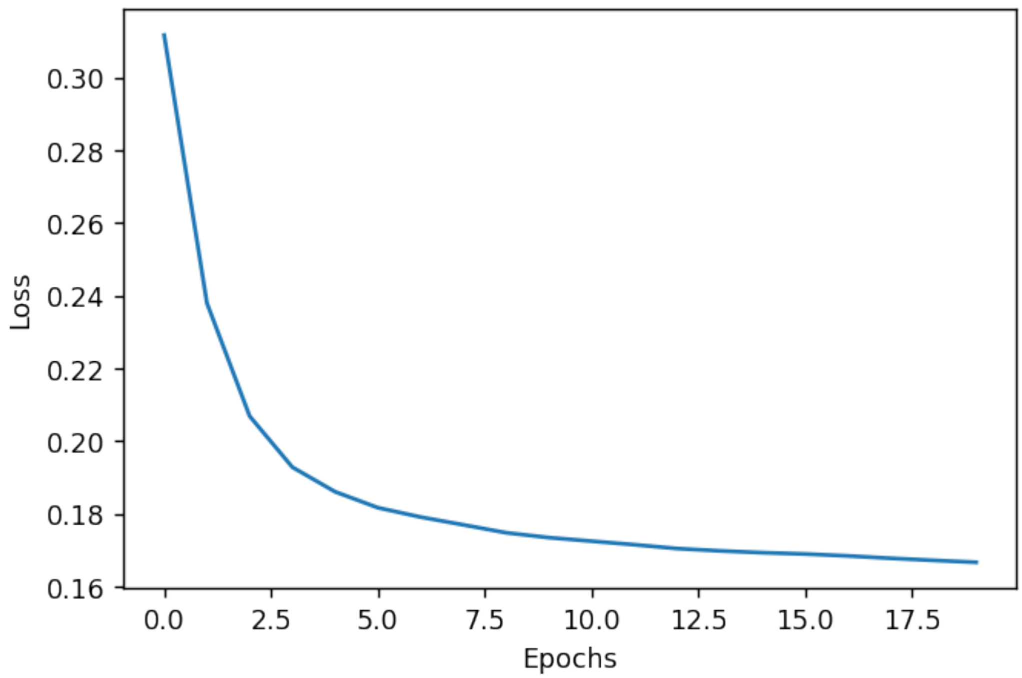
1.2 Test 2: test with 2 labels 1 and 9

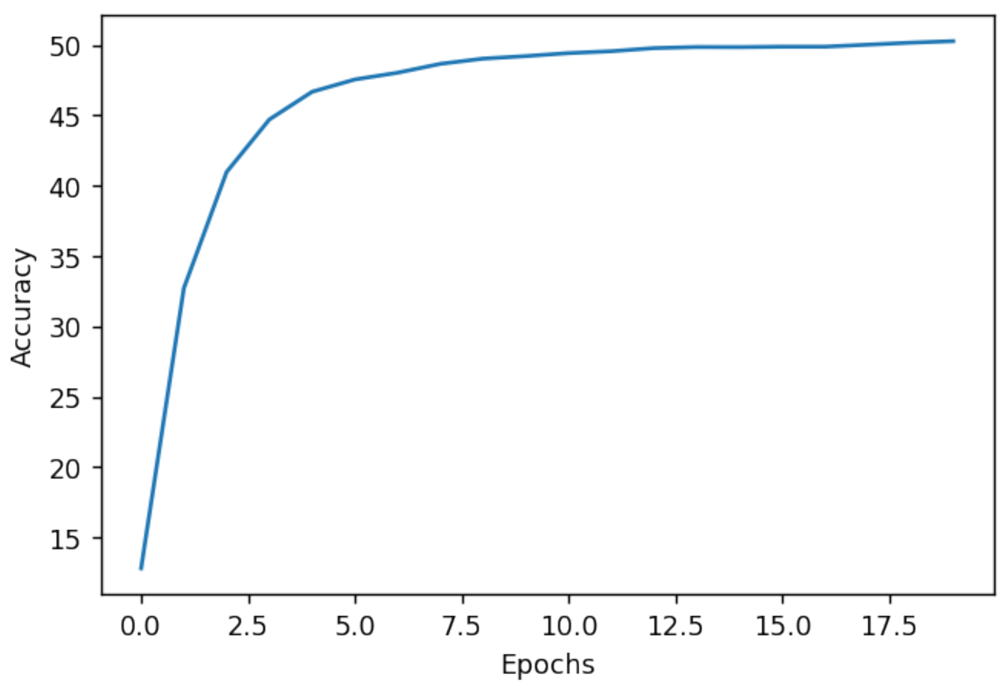
[264]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train[numpy.logical_or(y_train == 1, y_train == 9)]
y_train = y_train[numpy.logical_or(y_train == 1, y_train == 9)]

x_test = x_test[numpy.logical_or(y_test == 1, y_test == 9)]
y_test = y_test[numpy.logical_or(y_test == 1, y_test == 9)]

x_train = x_train/255
x_test = x_test/255
#reshape the data so it can fit the neural network model
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

from keras.utils import to_categorical
y_train = to_categorical(y_train, num_classes=10) #reshape into categorical␣
↪data 0 or 1
y_test = to_categorical(y_test, num_classes=10)

print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

```
(12691, 784) (12691, 10)
(2144, 784) (2144, 10)
```

[267]:
```
#input_size, hidden_size1, hidden_size2, output_size, learning_rate, batch, x,␣
↪y):
nn = NeuralNetwork(784, 256, 128, 10, 1e-6, 64, relu, relu_derivative, softmax,␣
↪x_train, y_train, x_test, y_test,  20)
predict, real = nn.train()
nn.loss_plot()
nn.accuracy_plot()
```

```
100%|          | 20/20 [00:33<00:00,  1.70s/it]
```





[268]:
```
predict, real = nn.test()
```

```
Accuracy: 50.373134328358205 %
```

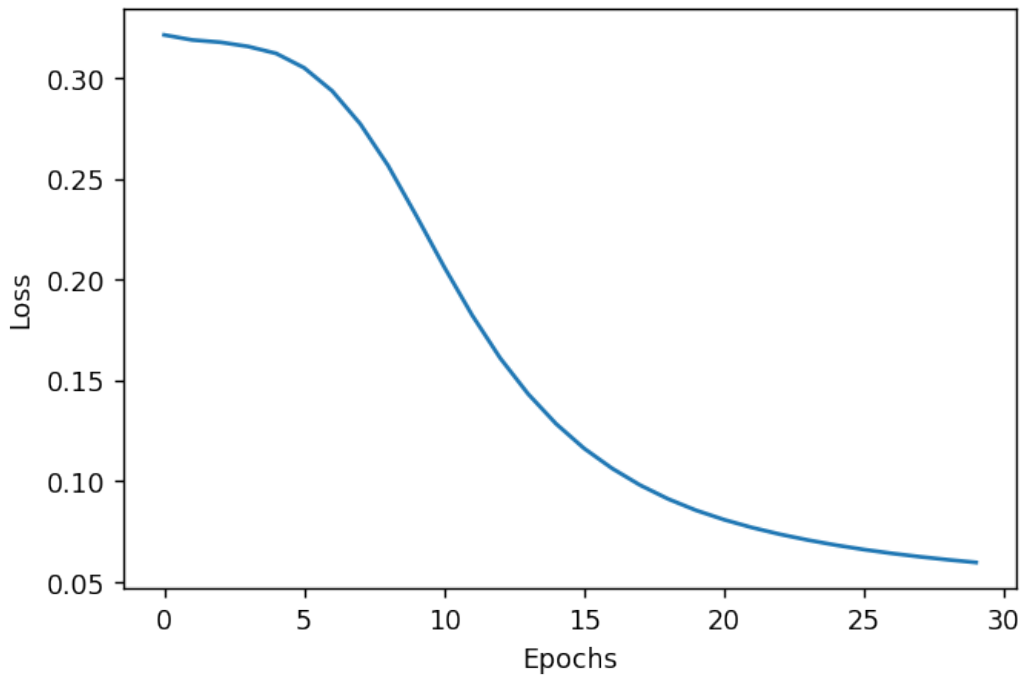
1.3 The dataset

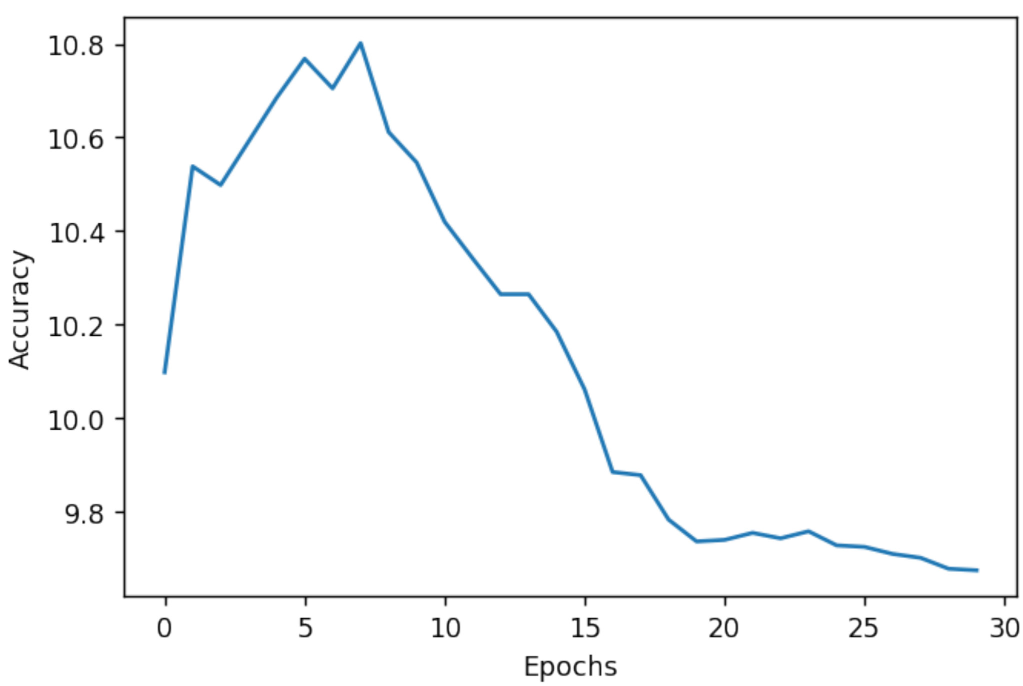
[269]:
```python
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train/255
x_test = x_test/255
#reshape the data so it can fit the neural network model
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)

from keras.utils import to_categorical
y_train = to_categorical(y_train, num_classes=10) #reshape into categorical␣
 ↪data 0 or 1
y_test = to_categorical(y_test, num_classes=10)

print(x_train.shape, y_train.shape)
print(x_test.shape, y_test.shape)
```

```
(60000, 784) (60000, 10)
(10000, 784) (10000, 10)
```

[272]:
```python
#input_size, hidden_size1, hidden_size2, output_size, learning_rate, batch, x,␣
 ↪y):
nn = NeuralNetwork(784, 256, 128, 10, 1e-6, 64, relu, relu_derivative, softmax,␣
 ↪x_train, y_train, x_test, y_test, num_epochs = 30)
predict, real = nn.train()
nn.loss_plot()
nn.accuracy_plot()
```

```
100%|          | 30/30 [02:33<00:00,  5.11s/it]
```





[273]:
```
predict, real = nn.test()
```

```
Accuracy: 9.120000000000001 %
```

1.4 Problem

It seems that my result has similar performance to random guess.

1.5 Learning

Observation 1: It performs so bad when I choose 50 hidden neurons, but improve so much better when I choose 256.

- How to choose the right number for the neurons in the hidden layer?
- Increasing the number of neurons can increase the capacity of the network and make it more capable of learning complex patterns in the data. However, having too many neurons can lead to overfitting, where the network becomes too specialized to the training data and is unable to generalize well to new data.
- Reason of choosing the factorial of 2: Easier computation.
- Resources: My Neural Network isn't working! What should I do?

Observation 2: Creating batches significantly improve my accuracy.

Observation 3: Learning Rate matters sooooooo much!

Since I am not able to figure out the error, I consult external resource and modify the code. I would appreciate suggestions to fix my code.

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:
```
!cp "./drive/My Drive/Deep Learning/Assignment1.ipynb" ./
!jupyter nbconvert --to pdf 'Assignment1.ipynb'
```

Attempt2

February 7, 2023

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt


def load_data(path):
    def one_hot(y):
        table = np.zeros((y.shape[0], 10))
        for i in range(y.shape[0]):
            table[i][int(y[i][0])] = 1
        return table

    def normalize(x):
        x = x / 255
        return x

    data = np.loadtxt('{}'.format(path), delimiter = ',', skiprows=1)
    return normalize(data[:,1:]),one_hot(data[:,:1])

X_train, y_train = load_data('mnist_train.csv')
X_test, y_test = load_data('mnist_test.csv')
```

[ ]:
```
X_train.shape[1]
```

[ ]: 784

[ ]:
```
class NeuralNetwork:
    def __init__(self, X, y, batch = 64, lr = 1e-3,  epochs = 10):
        self.input = X
        self.target = y
        self.batch = batch
        self.epochs = epochs
        self.lr = lr
        self.momentum = 0.7

        self.x = self.input[:self.batch] # batch input
        self.y = self.target[:self.batch] # batch target value
        self.loss = []
        self.acc = []
```

```
        self.init_weights()
        self.init_momentum()

    def init_weights(self):
        self.W1 = np.random.randn(self.input.shape[1],256)
        self.W2 = np.random.randn(self.W1.shape[1],128)
        self.W3 = np.random.randn(self.W2.shape[1],self.y.shape[1])

        self.b1 = np.random.randn(self.W1.shape[1],)
        self.b2 = np.random.randn(self.W2.shape[1],)
        self.b3 = np.random.randn(self.W3.shape[1],)

    def init_momentum(self):
        self.changeW3 = 0
        self.changeW2 = 0
        self.changeW1 = 0
        self.changeb3 = 0
        self.changeb2 = 0
        self.changeb1 = 0

    def ReLU(self, x):
        return np.maximum(0,x)

    def dReLU(self,x):
        return 1 * (x > 0)

    def softmax(self, z):
        z = z - np.max(z, axis = 1).reshape(z.shape[0],1)
        return np.exp(z) / np.sum(np.exp(z), axis = 1).reshape(z.shape[0],1)

    def shuffle(self):
        idx = [i for i in range(self.input.shape[0])]
        np.random.shuffle(idx)
        self.input = self.input[idx]
        self.target = self.target[idx]

    def feedforward(self):
        assert self.x.shape[1] == self.W1.shape[0]
        self.z1 = self.x.dot(self.W1) + self.b1
        self.a1 = self.ReLU(self.z1)

        assert self.a1.shape[1] == self.W2.shape[0]
        self.z2 = self.a1.dot(self.W2) + self.b2
        self.a2 = self.ReLU(self.z2)

        assert self.a2.shape[1] == self.W3.shape[0]
```

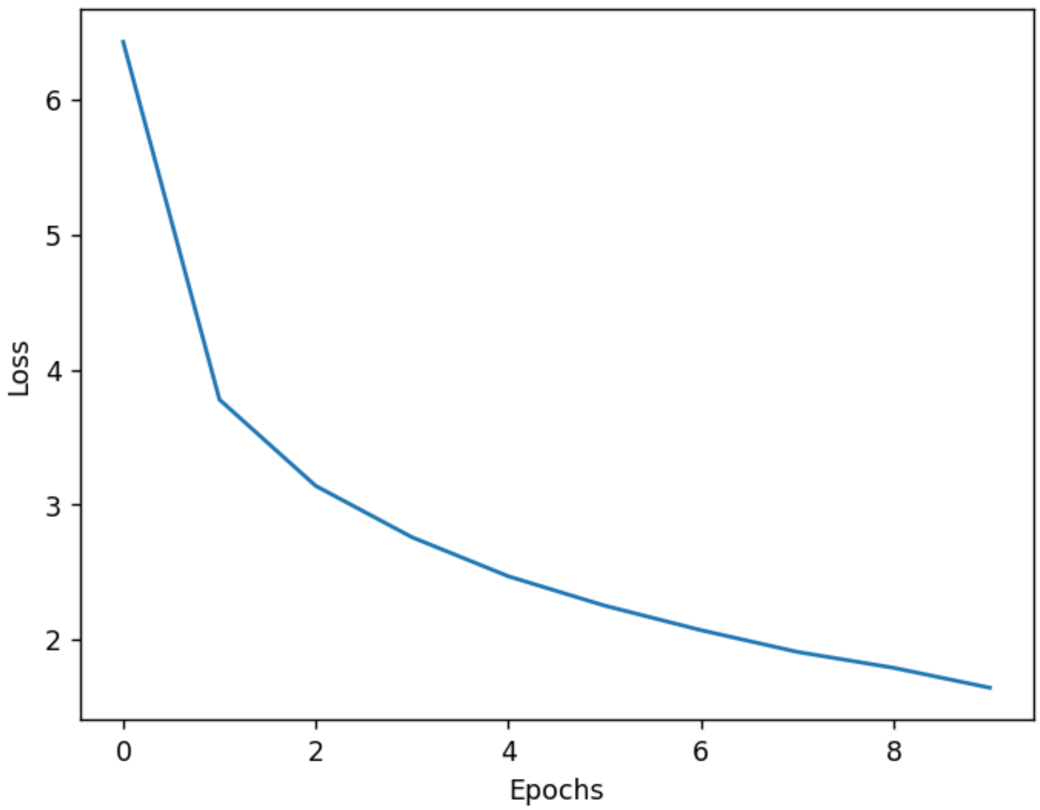
```python
        self.z3 = self.a2.dot(self.W3) + self.b3
        self.a3 = self.softmax(self.z3)
        self.error = self.a3 - self.y
        # self.error = self.y * np.log(self.a3)


    def backprop(self):
        dcost = (1/self.batch)*self.error

        DW3 = np.dot(dcost.T,self.a2).T
        DW2 = np.dot((np.dot((dcost),self.W3.T) * self.dReLU(self.z2)).T,self.
↪a1).T
        DW1 = np.dot((np.dot(np.dot((dcost),self.W3.T)*self.dReLU(self.z2),self.
↪W2.T)*self.dReLU(self.z1)).T,self.x).T

        db3 = np.sum(dcost,axis = 0)
        db2 = np.sum(np.dot((dcost),self.W3.T) * self.dReLU(self.z2),axis = 0)
        db1 = np.sum((np.dot(np.dot((dcost),self.W3.T)*self.dReLU(self.z2),self.
↪W2.T)*self.dReLU(self.z1)),axis = 0)

        assert DW3.shape == self.W3.shape
        assert DW2.shape == self.W2.shape
        assert DW1.shape == self.W1.shape

        assert db3.shape == self.b3.shape
        assert db2.shape == self.b2.shape
        assert db1.shape == self.b1.shape


        self.update_weight_with_momentum(DW3, DW2, DW1, db3, db2, db1)
        # self.W3 = self.W3 - self.lr * DW3
        # self.W2 = self.W2 - self.lr * DW2
        # self.W1 = self.W1 - self.lr * DW1

        # self.b3 = self.b3 - self.lr * db3
        # self.b2 = self.b2 - self.lr * db2
        # self.b1 = self.b1 - self.lr * db1

    def update_weight_with_momentum(self, DW3, DW2, DW1, db3, db2, db1):
        new_changeW3 = self.lr * DW3 + self.momentum * self.changeW3
        self.W3 = self.W3 - new_changeW3
        self.changeW3 = new_changeW3

        new_changeW2 = self.lr * DW2 + self.momentum * self.changeW2
        self.W2 = self.W2 - new_changeW2
        self.changeW2 = new_changeW2
```

```python
        new_changeW1 = self.lr * DW1 + self.momentum * self.changeW1
        self.W1 = self.W1 - new_changeW1
        self.changeW1 = new_changeW1

        new_changeb3 = self.lr * db3 + self.momentum * self.changeb3
        self.b3 = self.b3 - new_changeb3
        self.changeb3 = new_changeb3

        new_changeb2 = self.lr * db2 + self.momentum * self.changeb2
        self.b2 = self.b2 - new_changeb2
        self.changeb2 = new_changeb2

        new_changeb1 = self.lr * db1 + self.momentum * self.changeb1
        self.b1 = self.b1 - new_changeb1
        self.changeb1 = new_changeb1

        # print("changeb1", self.changeb1[:2])


    def train(self):
        for epoch in tqdm(range(self.epochs)):
            l = 0
            acc = 0
            self.shuffle()

            for batch in range(self.input.shape[0]//self.batch-1):
                start = batch*self.batch
                end = (batch+1)*self.batch
                self.x = self.input[start:end]
                self.y = self.target[start:end]
                self.feedforward()
                self.backprop()
                l+= self.cross_entropy_loss()
                acc+= np.count_nonzero(np.argmax(self.a3,axis=1) == np.argmax(self.y,axis=1)) / self.batch

            self.loss.append(l/(self.input.shape[0]//self.batch))
            self.acc.append(acc*100/(self.input.shape[0]//self.batch))

    def cross_entropy_loss(self):

        epsilon = 1e-15 # to avoid division by zero
        self.a3 = np.clip(self.a3, epsilon, 1 - epsilon)
        loss = -np.mean(np.sum(self.y * np.log(self.a3), axis=-1))
        return loss

    def plot(self):
```

```python
        plt.figure(dpi = 125)
        plt.plot(self.loss)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")

    def acc_plot(self):
        plt.figure(dpi = 125)
        plt.plot(self.acc)
        plt.xlabel("Epochs")
        plt.ylabel("Accuracy")

    def test(self,xtest,ytest):
        self.x = xtest
        self.y = ytest
        self.feedforward()
        acc = np.count_nonzero(np.argmax(self.a3,axis=1) == np.argmax(self.
 ↪y,axis=1)) / self.x.shape[0]
        print("Accuracy:", 100 * acc, "%")



NN = NeuralNetwork(X_train, y_train)
NN.train()
NN.plot()
NN.test(X_test,y_test)
```

```
100%|      | 10/10 [00:40<00:00,  4.04s/it]
```

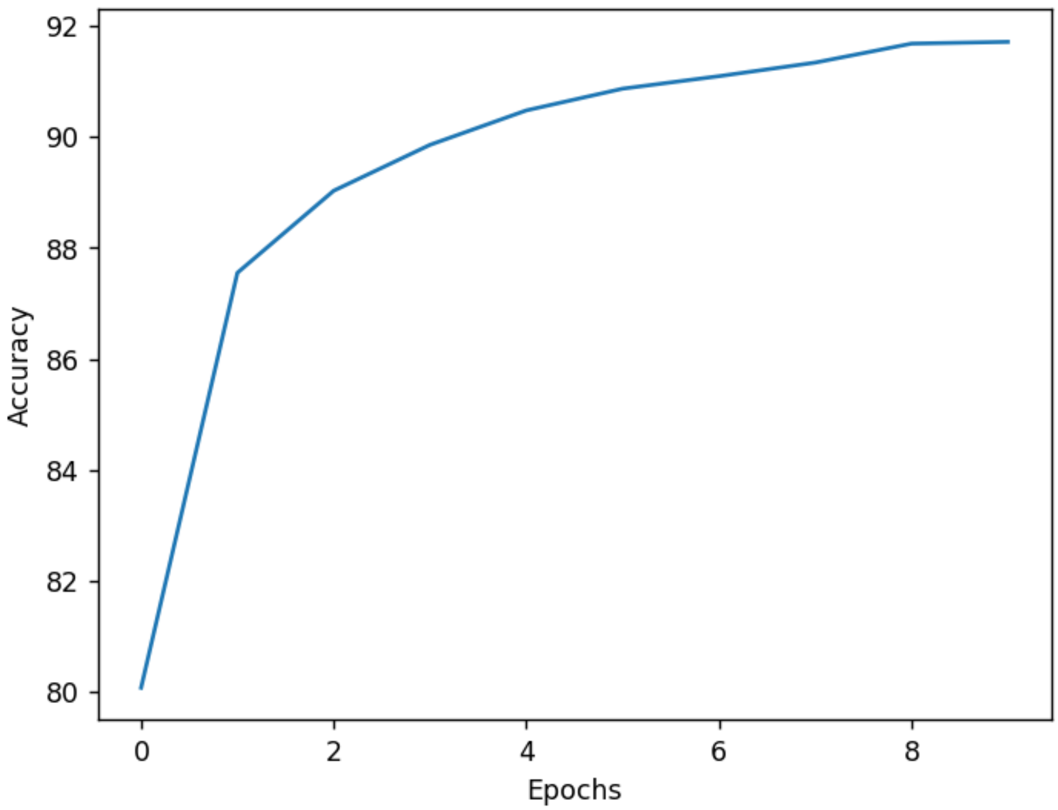
```
Accuracy: 90.34 %
```



Observation 4: -inf issue in categorical cross entropy loss: in the beggining, we have lots of 0 in the matrix. Since the log(0) isn't defined, we cannot calculate cross entropy loss. We resolve the problem through setting the min and max in the code: `np.clip(self.a3, epsilon, 1 - epsilon)`.

Observation 5: Setting momentum can both increase or decrease the accuracy for our model, but it always decreases the loss. In a good case, it can increase or decrease our accruacy by 2%. (ranging from 88% to 91 % accuracy). It might because since our model weight and bias is initialized randomly, the momentum can either drive our model to a good or bad diretion based on the initial setting. From our experiments momentum = 0.7 is the best for our model.

[ ]:
```
NN.acc_plot()
```



[ ]:
```
NN.test(X_test,y_test)
```

```
Accuracy: 90.34 %
```

[ ]:
```
```