Student Name: Mariem Ahmed

ECE421 - Winter 2021
Assignment 2:
Neural Networks

Due date: Monday, March 8, 2021

Submission: Submit both your report (a single PDF file) and all codes on Quercus.

Objectives:

The purpose of this assignment is to investigate the classification performance of neural networks. You will be implementing a neural network model using Numpy, followed by an implementation in Tensorflow. You are encouraged to look up TensorFlow APIs for useful utility functions, at: `https://www.tensorflow.org/api_docs/python/`.

In [ ]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import time
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'


# Load the data
def loadData():
    with np.load("notMNIST.npz") as data:
        Data, Target = data["images"], data["labels"]
        np.random.seed(521)
        randIndx = np.arange(len(Data))
        np.random.shuffle(randIndx)
        Data = Data[randIndx] / 255.0
        Target = Target[randIndx]
        trainData, trainTarget = Data[:10000], Target[:10000]
        validData, validTarget = Data[10000:16000], Target[10000:16000]
        testData, testTarget = Data[16000:], Target[16000:]
    return trainData, validData, testData, trainTarget, validTarget, tes
tTarget


# Implementation of a neural network using only Numpy - trained using gr
adient descent with momentum
def convertOneHot(trainTarget, validTarget, testTarget):
    newtrain = np.zeros((trainTarget.shape[0], 10))
    newvalid = np.zeros((validTarget.shape[0], 10))
    newtest = np.zeros((testTarget.shape[0], 10))

    for item in range(0, trainTarget.shape[0]):
        newtrain[item][trainTarget[item]] = 1
    for item in range(0, validTarget.shape[0]):
        newvalid[item][validTarget[item]] = 1
    for item in range(0, testTarget.shape[0]):
        newtest[item][testTarget[item]] = 1
    return newtrain, newvalid, newtest


def shuffle(trainData, trainTarget):
    np.random.seed(421)
    randIndx = np.arange(len(trainData))
    target = trainTarget
    np.random.shuffle(randIndx)
    data, target = trainData[randIndx], target[randIndx]
    return data, target
```

1. Neural Networks using Numpy [20 pts.]

1.1 Helper Functions [6 pt.]

1. ReLU(): This function will accept one argument and return Numpy array with the ReLU activation and the equation is given below. [0.5 pt]

```
ReLU(x) = max(x,0)
```

In [ ]:
```
#This function will accept one argument and return Numpy array with the
 ReLU activation
def relu(x):
    z = np.maximum(0, x)
    return z
```

1. softmax(): This function will accept one argument and return a Numpy array with the softmax activations of each of the inputs and the equation is shown below. [0.5 pt]

$$\sigma(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}, \quad j = 1, \cdots, K \text{ for K classes.}$$

Important Hint: In order to prevent overflow while computing exponentials, you should first subtract the maximum value of z from all its elements.

In [ ]:
```
#This function will accept one argument and return a Numpy array with th
e Softmax activations of each of the inputs
def softmax(x):
    a = np.max(x, axis=1)
    x = x - a.reshape(x.shape[0],1)
    sum_part = np.sum(np.exp(x), axis=1)
    z = np.exp(x) / sum_part.reshape(x.shape[0],1)
    return z
```

1. compute(): This function will accept 3 arguments: a weight matrix, an input vector, and a bias vector and return the product between the weights and input plus the biases (i.e. a prediction for a given layer). [0.5 pt]

In [ ]:
```
#Compute function will accept 3 arguments: a weight matrix, an input vec
tor, and a bias vector and return the product between the weights and in
put, plus the biases
def computeLayer(X, W, b):
    pose = np.transpose(b)
    x = np.dot(X, W) + pose
    return x
```

1. averageCE(): This function will accept two arguments, the targets (e.g. labels) and predictions - both are matrices of the same size. It will return a number, average the cross entropy loss for the dataset (i.e. training, validation, or test). For K classes, the formula is shown below. [0.5 pt]

$$\text{Average CE} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} \log\left(p_k^{(n)}\right)$$

Here, $y_k^{(n)}$ is the true one-hot label for sample n, p_k is the predicted class probability (i.e. softmax output for the k^{th} class) of sample n, and N is the number of examples.

In [ ]:
```
#This function will accept two arguments, the targets (e.g. labels) and
 predictions both are matrices of the same size.
#It will return a number, average the cross entropy loss for the dataset
def averageCE(target, prediction):
    part_Log = np.log(prediction)
    loss = (-1 / target.shape[0]) * np.sum(target * part_Log)
    return loss
```

1. gradCE(): This function will accept two arguments, the targets (i.e. labels y) and the input to the softmax function (i.e. o). It will return the gradient of the cross entropy loss with respect to the inputs to the softmax function: ∂L/∂o. **Show the analytical expression in your report.** [2 pt.]

In [ ]:
```
#This function will accept two arguments, the targets (i.e. labels y) an
d the input to the softmax function (i.e. o).
#It will return the gradient of the cross entropy loss with respect to t
he inputs to the softmax function
def gradCE(target, prediction):
    z = softmax(prediction)
    return z - target
```

Derivation of Gradient

$$\frac{\partial L}{\partial p_o} = \frac{\partial}{\partial p_o} \left(\frac{-1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_k^n log(p_k^n)\right)$$

$$= \left(\frac{-1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \frac{\partial\, y_k^n log(p_k^n)}{\partial\, p_o}\right)$$

$$= \left(\frac{-1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \frac{y_k^n}{p_k^n}\right)$$

1.2 Backpropagation Derivation [8 pts.]

To train the neural network, you will need to implement the backpropagation algorithm. For the neural network architecture outlined in the assignment description, derive the following analytical expressions and include them in your report:

Hints: The labels y have been one hot encoded. You will also need the derivative of the ReLU() function in order to backpropagate the gradient through the activation.

You may also wish to carry out your computations with the matrices transposed - this is also acceptable (although be careful when using the np.argmax function).

1. $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_o}$, the gradient of the loss with respect to the output layer weights. [1 pt.]
    - Shape: $(H \times 10)$, with H units

1. $$\frac{\partial L}{\partial W_o} = \frac{\partial L}{\partial x_o} \frac{\partial x_o}{\partial z_o} \frac{\partial z_o}{\partial W_o}$$

$$\frac{\partial L}{\partial x_o} = (\frac{-1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \frac{y_k^n}{p_k^n})$$

$$\frac{\partial x_o}{\partial z_o} = \begin{cases} if\ i = j \quad \frac{\partial x_{oi}}{\partial z_{oj}} = \frac{e^{zi} \sum_{k=1}^{K} e^{zk} - e^{zi} e^{zj}}{(\sum_{k=1}^{K} e^{zk})^2} = \frac{e^{zi}}{\sum_{k=1}^{K} e^{zk}} \frac{\sum_{k=1}^{K} e^{zk} - e^{zj}}{\sum_{k=1}^{K} e^{zk}} = p_i(1 - p_i) \\ if\ i \neq j \quad \frac{\partial x_{oi}}{\partial z_{oj}} = \frac{0 - e^{zi} e^{zj}}{\sum_{k=1}^{K} e^{zk}} = \frac{e^{zi}}{\sum_{k=1}^{K} e^{zk}} \frac{e^{zj}}{\sum_{k=1}^{K} e^{zk}} = -p_i p_j \end{cases}$$

$$\frac{\partial z_o}{\partial W_o} = \frac{\partial x_h W_o + b_o}{\partial W_o} = x_h$$

$$\implies \frac{\partial L}{\partial W_o} = (\frac{-1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \frac{y_k^n}{p_k^n}) \begin{cases} p_i(1 - p_i) & if\ i = j \\ -p_i p_j & if\ i \neq j \end{cases} x_h$$

2. $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_o}$, the gradient of the loss with respect to the output layer biases. [1 pt.]
    - Shape: (1×10)

$$2.\ \frac{\partial L}{\partial b_o} = \frac{\partial L}{\partial x_o} \frac{\partial x_o}{\partial z_o} \frac{\partial z_o}{\partial W_o}$$

$$\frac{\partial z_o}{\partial b_o} = \frac{\partial x_h W_o + b_o}{\partial W_o} = 1$$

3. $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h}$, the gradient of the loss with respect to the hidden layer weights. [1 pt.]

- Shape: $(F \times H)$, with F features, H units

3. $$\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial x_o} \frac{\partial x_o}{\partial z_o} \frac{\partial z_o}{\partial x_h} \frac{\partial x_h}{\partial z_h} \frac{\partial z_h}{\partial W_h}$$

$$\frac{\partial z_o}{\partial x_h} = \frac{\partial x_h W_o + b_o}{\partial x_h} = W_o$$

$$\frac{\partial x_h}{\partial z_h} = \begin{cases} 0 & if \ z_h \leq 0 \\ 1 & if \ z_h > 0 \end{cases} = (x_h \geq 0)$$

$$\frac{\partial z_h}{\partial W_h} = \frac{\partial x_{in} W_h + b_h}{\partial W_h} = x_{in}$$

4. $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_h}$, the gradient of the loss with respect to the hidden layer biases. [1 pt.]

- Shape: $(1 \times H)$, with H units.

4. $$\frac{\partial L}{\partial b_h} = \frac{\partial L}{\partial x_o} \frac{\partial x_o}{\partial z_o} \frac{\partial z_o}{\partial x_h} \frac{\partial x_h}{\partial z_h} \frac{\partial z_h}{\partial b_h}$$

$$\frac{\partial z_h}{\partial b_h} = \frac{\partial x_{in} W_h + b_h}{\partial b_h} = 1$$

1.3 Learning [6 pts.]

Construct the neural network and train it for 200 epochs with a hidden unit size of $H = 1000$. First, initialize your weight matrices following the Xaiver initialization scheme (zero-mean Gaussians with variance $\frac{2}{\text{units in}+\text{units out}}$) and your bias vectors to zero, each with the shapes as outlined in section 1.2. Using these matrices, compute a forward pass of the training data and then, using the gradients derived in section 1.2, implement the backpropagation algorithm to update all of the network's weights and biases. The optimization technique to be used for backpropagation will be Gradient Descent with momentum and the equation is shown below.

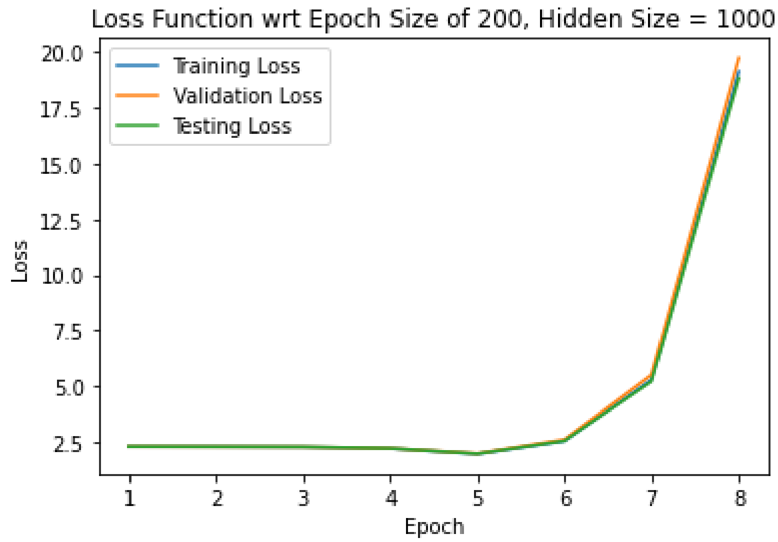
$$\boldsymbol{\nu}_{\text{new}} \leftarrow \gamma \boldsymbol{\nu}_{\text{old}} + \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boldsymbol{\nu}_{\text{new}}$$

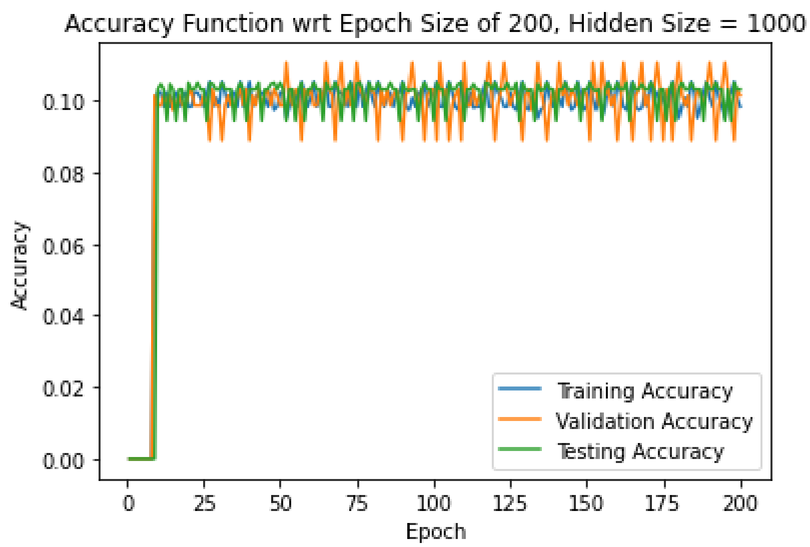
For the $\boldsymbol{\nu}$ matrices, initialize them to the same size as the hidden and output layer weight matrix sizes, with a very small value (e.g. 10^{-5}). Additionally, initialize your γ values to values slightly less than 1 (e.g. 0.9 or 0.99) and set $\alpha = 0.1$ for the average loss. (Note that you need to scale the learning rate if you are using the total loss).

Plot the training and validation loss in one figure, and the training and validation accuracy curves in a second figure and include them in your report. For the accuracy metric, the `np.argmax()` function will be helpful.

Answer:

Below is the plot for Loss and Accuracy. For Loss plot, it converges, no noise and smooth but for Accuracy plot, it started smooth at the beginning then it starts to be noisy at the end this might be due to the very small value of alpha.





In [ ]:
```
#The code below combining 1.2 and 1.3 question to get the Output and Hid
den Layer
#This function will get the accuracy metric using the np.argmax() functi
on
def get_Accuracy(y_OneHot, target):
    y_OneHot = np.argmax(y_OneHot)
    y_OneHot = np.array(y_OneHot)
    accuracy = np.where(y_OneHot == target, 1, 0).sum()
    acc = accuracy/target.shape[0]
    return acc

#This fucntion computes Forward Propagation using 4 arguments: input, ou
tput_weight, output_bias, hidden_weight, hidden_bias
def compute_forward_Propagation(input, output_weight, output_bias, hidde
n_weight, hidden_bias):
    input = computeLayer(input, hidden_weight, hidden_bias)
    input = relu(input)
    input = computeLayer(input, output_weight, output_bias)
    return input

#Using these two website below to come up with some idea of how to imple
ment Forward and Backward Propagation:
#https://towardsdatascience.com/building-a-neural-network-with-a-single-
hidden-layer-using-numpy-923be1180dbf
#https://techcommunity.microsoft.com/t5/educator-developer-blog/how-to-i
mplement-the-backpropagation-using-python-and-numpy/ba-p/378895

#Using same code from Assignemt #1
def train_model(hidden_units, num_epoch=200):
    trainData, validData, testData, trainTarget, validTarget, testTarget
= loadData()
    trainTargetOneHot, validTargetOneHot, tesTargetOneHot = convertOneHo
t(trainTarget,validTarget,testTarget)
    trainData = trainData.reshape(-1, 784)
    validData = validData.reshape(-1, 784)
    testData = testData.reshape(-1, 784)

    # Initialize Hyperparameters
    γ = 0.99
    alpha = 10e-5

    # Initialize Weights
    units_in = trainData.shape[1]
    units_out = hidden_units
    K = 10

    loss = [0] * num_epoch
    accuracy = [0] * num_epoch
    loss_valid = [0] * num_epoch
    accuracy_valid = [0] * num_epoch
    loss_test = [0] * num_epoch
    accuracy_test = [0] * num_epoch

    #Output Layer Weights and Biases
    output_weight = np.random.randn(units_out, K) * 2 / (units_out + K)
    units_output_weight = np.ones((units_out, K)) * alpha
```

```
    output_bias = np.random.randn(K, 1) * 2 / (K + 1)
    units_output_bias = np.ones((K, 1)) * alpha

    #Hidden Layer Weights and Biases
    hidden_weight = np.random.randn(units_in, units_out) * 2 / (units_in
+ units_out)
    units_hidden_weight = np.ones((units_in, units_out)) * alpha
    hidden_bias = np.random.randn(units_out, 1) * 2 / (units_out + 1)
    units_hidden_bias = np.ones((units_out, 1)) * alpha

    for epoch in range(num_epoch):

        #Forward Propagation
        pred_out = computeLayer(trainData, hidden_weight, hidden_bias)
        Out_hidden = relu(pred_out)
        prediction = computeLayer(Out_hidden, output_weight, output_bias
)

        pred_valid = compute_forward_Propagation(validData, output_weigh
t, output_bias, hidden_weight, hidden_bias)
        pred_test = compute_forward_Propagation(testData, output_weight,
output_bias, hidden_weight, hidden_bias)

        prediction = softmax(prediction)
        prediction_valid = softmax(pred_valid)
        prediction_test = softmax(pred_test)

        #Backward Propagation
        grad_loss = gradCE(trainTargetOneHot, prediction)
        grad_output_weight = np.dot(np.transpose(Out_hidden), grad_loss)
        grad_output_bias = np.transpose(sum(grad_loss)).reshape(K, 1)

        pro_dot = np.dot(grad_loss, np.transpose(output_weight))
        grad_hidden_weight = np.dot(np.transpose(trainData), np.where(Ou
t_hidden > 0, 1, 0) * pro_dot)
        grad_hidden_bias = sum(np.where(Out_hidden > 0, 1, 0) * np.dot(g
rad_loss, np.transpose(output_weight))).reshape(hidden_units, 1)


        # Update Parameters of Output Layer for Weights and Biases
        units_output_weight = γ * units_output_weight + alpha * grad_outp
ut_weight
        output_weight -= units_output_weight
        unitsoutput_bias = γ * units_output_bias + alpha * grad_output_bi
as
        output_bias -= units_output_bias

        #Update Parameters of Hidden Layer for Weights and Biases
        units_hidden_weight = γ * units_hidden_weight + alpha * grad_hidd
en_weight
        hidden_weight -= units_hidden_weight
        units_hidden_bias = γ * units_hidden_bias + alpha * grad_hidden_b
ias
        hidden_bias -= units_hidden_bias

        #Loss Function with respect to Epoch Size
        loss[epoch] = averageCE(trainTargetOneHot, prediction)
```

```
        loss_valid[epoch] = averageCE(validTargetOneHot, prediction_vali
d)
        loss_test[epoch] = averageCE(tesTargetOneHot, prediction_test)

        #Accuracy Function with respect to Epoch Size
        accuracy[epoch] = get_Accuracy(prediction, trainTarget)
        accuracy_valid[epoch] = get_Accuracy(prediction_valid, validTarg
et)
        accuracy_test[epoch] = get_Accuracy(prediction_test, testTarget)

    # Plot Loss Function
    plot_epoch = [i for i in range(1, num_epoch + 1)]
    plt.figure()
    plt.title('Loss Function wrt Epoch Size of 200, Hidden Size = '+str(
hidden_units))
    plt.plot(plot_epoch, loss, label='Training Loss')
    plt.plot(plot_epoch, loss_valid, label='Validation Loss')
    plt.plot(plot_epoch, loss_test, label='Testing Loss')
    plt.ylabel("Loss")
    plt.xlabel("Epoch")
    plt.legend()
    plt.show()

    # Plot Accuracy Function
    plt.figure()
    plt.title('Accuracy Function wrt Epoch Size of 200, Hidden Size = '+
str(hidden_units))
    plt.plot(plot_epoch, accuracy, label='Training Accuracy')
    plt.plot(plot_epoch, accuracy_valid, label='Validation Accuracy')
    plt.plot(plot_epoch, accuracy_test, label='Testing Accuracy')
    plt.ylabel("Accuracy")
    plt.xlabel("Epoch")
    plt.legend()
    plt.show()

if __name__ == "__main__":
    hidden_size = 1000
    epoch = 200
    train_model(hidden_size)
```