# Homework Assignment - 2

**Name:** Sagar Patel

**NETID:** sp5894

---

## Question 1

*Analyzing gradient descent.* Consider a simple functon having two weight variables:

$L(w_1, w_2) = 0.5(aw_1^2 + bw_2^2)$

### 1.a

Write down the gradient $\nabla L(w)$, and using this, derive the weights $w^*$ that achieve the minimum value of $L$.

**Solution:**

Given that $\nabla L(w_1, w_2) = 0.5(aw_1^2 + bw_2^2)$, we can differentiate $L(w_1, w_2)$ with respect to $w_1$ and $w_2$ (Gradient)

$\frac{\partial L(w_1,w_2)}{\partial w_1} = aw_1$ and $\frac{\partial L(w_1,w_2)}{\partial w_2} = bw_2$

Therefore, the Gradient of $\nabla L(w_1, w_2) = < aw_1, bw_2 >$

In order to get the minima of the gradient, we equate it to zero $\Rightarrow \nabla L(w_1, w_2) = 0$

$\Rightarrow aw_1 = 0$ and $bw_2 = 0$

This means that at minima, $w_1 = w_2 = 0$

Therefore, $w^* =< 0, 0 >$

### 1.b

Instead of simply writing down the optimal weights, let's now try to optimize $L$ using gradient descent. Starting from some randomly chosen (non-zero) initialization point $w_1(0), w_2(0)$, write down the gradient descent updates. Show that the updates have the form:

$w_1(t + 1) = \rho_1 w_1(t),$ and $w_2(t + 1) = \rho_2 w_2(t)$

where $w_i(t)$ represent the weights at the t^{th} iteration. Derive the expressions for $\rho_1$ and $\rho_2$ in terms of $a, b$, and the learning rate.

**Solution:**

We know that, $w(t + 1) = w(t) - \eta\nabla L(w(t))$ (where $\eta$ is the Learning Rate)

If we consider $w_1$ and $w_2$, the equations can be -

$w_1(t+1) = w_1(t) - \eta \nabla L(w_1(t)) \ldots (1)$

$w_2(t+1) = w_2(t) - \eta \nabla L(w_2(t)) \ldots (2)$

Now, we know that $\nabla L(w_1(t)) = aw_1(t)$ and $\nabla L(w_2(t)) = bw_2(t)$. Substituting these in equations (1) and (2), we get -

$w_1(t+1) = w_1(t) - \eta a w_1(t) \ldots (3)$

$w_2(t+1) = w_2(t) - \eta b w_2(t) \ldots (4)$

For a non-zero element, $w_1(0)$ and $w_2(0) = 1$, we have -

$w_1(1) = 1 - \eta a \ldots (5)$

$w_2(1) = 1 - \eta b \ldots (6)$

We can notice that this generalizes well for other values of $t$ -

$w_1(t+1) = w_1(t)(1 - \eta a) \Rightarrow w_1(t+1) = (1 - \eta a)w_1(t)$

$w_2(t+1) = w_2(t)(1 - \eta b) \Rightarrow w_2(t+1) = (1 - \eta b)w_2(t)$

Now this resembles the form -

$w_1(t+1) = \rho_1 w_1(t)$ and $w_2(t+1) = \rho_2 w_2(t)$

Therefore, this indicates that $\rho_1 = (1 - \eta a)$ and $\rho_2 = (1 - \eta b)$

### 1.c

Under what values of the learning rate does gradient descent converge to the correct minimum? Under what values does it not?

**Solution:**

From $\rho_1 = (1 - \eta a)$ and $\rho_2 = (1 - \eta b)$, we can understand that the values of $\rho_1$ and $\rho_2$ should have a range [-1,1]

The reason why we have the range varying from -1 to +1 is because we want to bring the negative and positive numbers clsoer to zero respectively.

$1 < (1 - \eta a) < +1$

$1 < (1 - \eta b) < +1$

$\Rightarrow 0 < \eta < 2/a \ldots (1)$

$\Rightarrow 0 < \eta < 2/b \ldots (2)$

The values are said to converge on any point between the said range. However, that does not mean they converge at zero because if $(1 - \eta a)$ at $\eta = 0$ will not change the value of $w_1(t)$

Therefore, $0 < \eta < min(2/a, 2/b)$

### 1.d

Provide a scenario under which the convergence rate of gradient descent is very slow.

(Hint: consider the case where $a/b$ is a very large ratio)

**Solution:**

Given that,

$\Rightarrow 0 < \eta < 2/a$

$\Rightarrow 0 < \eta < 2/b$

We know that the largest value of $\eta$ is going to be "slightly" lesser than the minimum value of $2/a$ and $2/b$. In the case where $a/b$ is a very large ratio, the value of $\eta$ would be $2/a$.

This number would be very very tiny and that would lead to a slower convergence rate of the gradient descent.

## Question 2

Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other cloud service of your choice) and complete the missing items. Save your finished notebook in PDF format and upload along with your answers to the above theory questions in a single PDF.

**Solution:**

Link to the Notebook

**Name:** Sagar Patel

**NETID:** sp5894

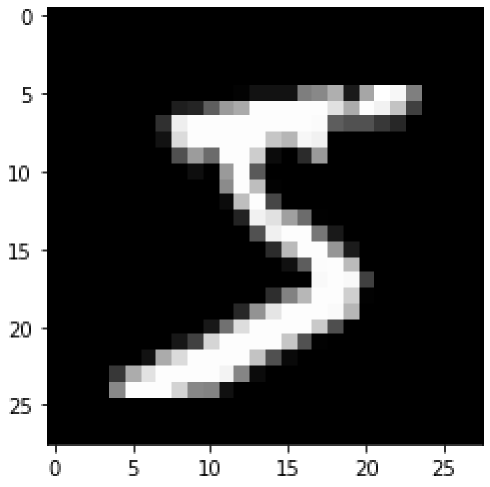
In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

```
import tensorflow as tf
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data(path="mnist.r

plt.imshow(x_train[0],cmap='gray');
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

  x = np.clip(x, -500, 500) # We get an overflow warning without this
```

```
    return np.where(
      x >= 0,
      1 / (1 + np.exp(-x)),
      np.exp(x) / (1 + np.exp(x))
    )

def dsigmoid(x): # Derivative of sigmoid
    return sigmoid(x) * (1 - sigmoid(x))

def softmax(x):
    # Numerically stable softmax based on (same source as sigmoid)
    # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/
    b = x.max()
    y = np.exp(x - b)
    return y / y.sum()

def cross_entropy_loss(y, yHat):
    return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max):
    # x: integer to convert to one hot encoding
    # max: the size of the one hot encoded array
    result = np.zeros(10)
    result[x] = 1
    return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance $1/\max(n_{in}, n_{out})$.

```
import math

# Initialize weights of each layer with a normal distribution of mean 0 and
# standard deviation 1/sqrt(n), where n is the number of inputs.
# This means the weighted input will be a random variable itself with mean
# 0 and standard deviation close to 1 (if biases are initialized as 0, standard
# deviation will be exactly 1)

from numpy.random import default_rng

rng = default_rng(80085)

# Q1. Fill initialization code here.
# ...
size = [784*32, 32*32, 32*10]
b = [32, 32, 10]
mu = [0,0,0]
sigma = [1/math.sqrt(784), 1/math.sqrt(32), 1/math.sqrt(32)]

w1 = np.random.normal(loc= mu[0], scale= sigma[0], size= 784*32).reshape(784,32)
w2 = np.random.normal(loc= mu[1], scale= sigma[1], size= 32*32).reshape(32,32)
w3 = np.random.normal(loc= mu[1], scale= sigma[1], size= 32*10).reshape(32,10)

weights = [w1, w2, w3]
biases = [np.zeros((1,b[i])) for i in range(3)]
```

```
x_train.shape
```

```
(60000, 28, 28)
```

```
y_train[0]
```

```
5
```

Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test accuracy, it should be somewhere around 1/10 (i.e., a random guess).

```
def feed_forward_sample(sample, y):
  """ Forward pass through the neural network.
    Inputs:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: the cross entropy loss andmost likely class of "sample"
  """
  # Q2. Fill code here.
  # ...
  L1 = sigmoid(np.dot(sample, weights[0]) + biases[0])
  L2 = sigmoid(np.dot(L1, weights[1]) + biases[1])
  output = softmax(np.dot(L2, weights[2]) + biases[2])
  one_hot_guess = integer_to_one_hot(np.argmax(output), 10)

  y = integer_to_one_hot(y, 10)
  loss = cross_entropy_loss(y, output)
  return loss, one_hot_guess
  return loss, one_hot_guess


def feed_forward_dataset(x, y):
  losses = np.empty(x.shape[0])
  one_hot_guesses = np.empty((x.shape[0], 10))



  # ...
  # Q2. Fill code here to calculate losses, one_hot_guesses
  # ...
  x = x.reshape(-1, 784)
  for i in range(len(x)):
    l, h = feed_forward_sample(x[i], y[i])
    losses[i] = l
    one_hot_guesses[i] = h

  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1

  correct_guesses = np.sum(y_one_hot * one_hot_guesses)
  correct_guess_percent = format((correct_guesses / y.shape[0]) * 100, ".2f")

  print("\nAverage loss:", np.round(np.average(losses), decimals=2))
  print("Accuracy (# of correct guesses):", correct_guesses, "/", y.shape[0], "(", cor

def feed_forward_training_data():
  print("Feeding forward all training data...")
  feed_forward_dataset(x_train, y_train)
  print("")

def feed_forward_test_data():
  print("Feeding forward all test data...")
  feed_forward_dataset(x_test, y_test)
  print("")

feed_forward_test_data()
```

```
    Feeding forward all test data...

    Average loss: 1.0
    Accuracy (# of correct guesses): 1132.0 / 10000 ( 11.32 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

```
def train_one_sample(sample, y, learning_rate=0.003):
    a = sample.flatten()

    # We will store each layer's activations to calculate gradient
    activations = []

    # Forward pass

    # Q3. This should be the same as what you did in feed_forward_sample above.
    # ...
    A1 = np.dot(a,weights[0]) + biases[0]
    Z1 = sigmoid(A1)

    A2 = np.dot(Z1,weights[1])+biases[1]
    Z2 = sigmoid(A2)

    A3 = np.dot(Z2,weights[2])+biases[2]
    Z3 = softmax(A3)

    one_hot_guess = integer_to_one_hot(np.argmax(Z3),10)
    y = integer_to_one_hot(y,10)
    loss = cross_entropy_loss(y,Z3)

    activations.append(Z1)
    activations.append(Z2)
    activations.append(Z3)

    # Backward pass

    # Q3. Implement backpropagation by backward-stepping gradients through each layer.
    # You may need to be careful to make sure your Jacobian matrices are the right shape
    # At the end, you should get two vectors: weight_gradients and bias_gradients.
    # ...
    dZ3 = -1 * (one_hot_guess - (one_hot_guess*Z3))
    Z2 = Z2.reshape(1,32)
    dZ3 = dZ3.reshape(1,10)
    dW3 = np.dot(Z2.T,dZ3)
    dB3 = dZ3

    dZ2 = dsigmoid(Z2)*np.dot(dZ3,Z3.T)
    A1 = A1.reshape(1,32)
    dZ2 = dZ2.reshape(1,32)
    dW2 = np.dot(A1.T,dZ2)
    dB2 = dZ2

    dZ1 = dsigmoid(Z1)*np.dot(dZ2,Z2.T)
    a = a.reshape(1,784)
    dZ1 = dZ1.reshape(1,32)
    dW1 = np.dot(a.T,dZ1)
    dB1 = dZ1

    weight_gradients = [dW1,dW2,dW3]
```

```
  bias_gradients = [dB1,dB2,dB3]
  for i in range(0,3):
    # Update weights & biases based on your calculated gradient
    weights[i] -= weight_gradients[i] * learning_rate
    biases[i] -= bias_gradients[i].flatten() * learning_rate
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

```
def train_one_epoch(learning_rate=0.003):
  print("Training for one epoch over the training dataset...")

  # Q4. Write the training loop over the epoch here.
  # ...
  for i in range(len(x_train)):
    train_one_sample(x_train[i],y_train[i])
  print("Finished training.\n")


feed_forward_test_data()

def test_and_train():
  train_one_epoch()
  feed_forward_test_data()

for i in range(3):
  print('Epoch ', i)
  test_and_train()
```

```
Feeding forward all test data...

Average loss: 1.0
Accuracy (# of correct guesses): 1010.0 / 10000 ( 10.10 %)

Epoch  0
Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 1.0
Accuracy (# of correct guesses): 1010.0 / 10000 ( 10.10 %)

Epoch  1
Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 1.0
```

```
Accuracy (# of correct guesses): 1010.0 / 10000 ( 10.10 %)

Epoch  2
Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 1.0
Accuracy (# of correct guesses): 1010.0 / 10000 ( 10.10 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~85% accuracy after 3 epochs.