Homework 2

Monica Yao

1.

Since this problem is using sigmoid and relu as its activation functions, I defined my functions the same way as we did in class (from Lecture 7). I also defined the loss function and the derivative of the loss function.

I also used multiple functions to do forward propagation, which will calculate the output this neural network gives. The param dictionary is for all the inputs that we have. The functions I wrote will also puts every weighted sum and output of each neuron into the dictionary, and spit out a new one along with the output.

In [1]:

```
import numpy as np

def sigmoid_np(a):
    return 1/(1+np.exp(-a))

def sigmoid_np_prime(a):
    return sigmoid_np(a)*(1-sigmoid_np(a))

def relu_np(z):
    return np.maximum(0 ,z)

def relu_np_prime(z):
    return np.where(z <= 0, 0, 1)

def weightedsum_np(A, W, b):
    return np.matmul(W, A) + b

def loss_function(y, yHat):
    return -(y*np.log(yHat)+(1-y)*np.log(1-yHat))

def loss_function_prime(y, yHat):
    return -y/yHat + 1/(1-yHat) - y/(1-yHat)

params = {
    'x': np.array([[1.0], [-2.0], [3.0]]),
    'W1': np.array([[1.0, 2.0, 3.0]]),
    'b1': np.array([-4.0]),
    'W2': np.array([[2.0], [8.0]]),
    'b2': np.array([[6.0], [5.0]]),
    'W3': np.array([[8.0, 2.0]]),
    'b3': np.array([-120.0]),
}

# I calculate the output of layer1, which becomes the input of our next layer
# This uses activation function relu
def firstLayer(params):
    params['layer1z'] = weightedsum_np(params['x'], params['W1'], params['b1'])
    params['layer1a'] = relu_np(params['layer1z'])
    return params

# I calculate the output of layer2, which becomes the input of our next layer
# This uses activation function relu
def secondLayer(params):
    params = firstLayer(params)
    params['layer2z'] = weightedsum_np(params['layer1a'], params['W2'], params['b2'])
    params['layer2a'] = relu_np(params['layer2z'])
    return params

# I calculate the output
# This uses activation function sigmoid
def outputLayer(params):
    params = secondLayer(params)
    params['layer3z'] = weightedsum_np(params['layer2a'], params['W3'], params['b3'])
    params['layer3a'] = sigmoid_np(params['layer3z'])
    return params, params['layer3a']
```

Backpropagation

We use backpropagation where we will use the chain rule to find the partial derivative of the loss function with respect to the weight from the first input to the first neuron in the first layer.

In [2]:

```
# I declare variables for a new dictionary that has all the weights and outputs of each neuron, an
d the final output
newParams, output = outputLayer(params)

# Calculating error at the last layer
dJ_dlayer3a = loss_function_prime(1, output)
dJ_dlayer3z = np.matmul(dJ_dlayer3a, sigmoid_np_prime(newParams['layer3z']))

# Calculating the partial derivative of the layer with repect to the earlier layer
dlayer3z_dlayer2z = relu_np_prime(newParams['layer2z'])
dlayer2z_dlayer1z = relu_np_prime(newParams['layer1z'])

# Using chain rule to calculate the error at the last layer
dJ_dlayer2z = np.multiply(np.matmul(newParams['W3'].transpose(), dJ_dlayer3z), dlayer3z_dlayer2z)
dJ_dlayer1z = np.multiply(np.matmul(newParams['W2'].transpose(), dJ_dlayer2z), dlayer2z_dlayer1z)

# Calculating the error for the weights at layer 1 using the error at that layer
dJ_dW1 = np.matmul(params['x'], dJ_dlayer1z)

print('Error for the weights at layer one')
print(f'dJ_dW1: {dJ_dW1}')
print('Error for the weights at first neuron at layer onelayer one')
print(f'dJ_dW1[0]: {dJ_dW1[0]}')
```

```
Error for the weights at layer one
dJ_dW1: [[ -3.8144935 ]
 [  7.62898701]
 [-11.44348051]]
Error for the weights at first neuron at layer onelayer one
dJ_dW1[0]: [-3.8144935]
```

Numerical Differentiation

We now use a numerical differentiation to make sure that our calculations are correct. We calculate the actual value of $J(y, \hat{y})$ then find the partial derivative $\frac{\partial J}{w_1^{[1][1]}}$ by the defition of the derivative by changing the weight by a small amount epsilon.

In [3]:

```
# Calculate the output yHat then we call our loss function to find the loss
newParams, output = outputLayer(params)
j_out = loss_function(1, output)

# Clone params, add epsilon to the first weight of the first neuron of the first layer
params_epsilon = dict(newParams)
epsilon = .000001
newParams['W1'][0][0] += epsilon

# Calculating the output and loss with new weight
_, epsilon_output = outputLayer(params_epsilon)
jepsilon = loss_function(1, epsilon_output)

# Calculating the partial derivative using definition
dJ_dw1 = (jepsilon-j_out)/epsilon

print('Error for the weights at first neuron at layer onelayer one')
print(f'dJ/dw1: {dJ_dw1}')
```

```
Error for the weights at first neuron at layer onelayer one
dJ/dw1: [[-3.81443975]]
```

The two partial derivatives match up!

They are both around $\frac{\partial J}{w_1{}^{[1]}{}_{[1]}} = -3.814$.

2

a) For $tan(x)$, a good activation function would be a linear or cubic function. $tan(x)$ has range from $-inf$ to inf, so a good activation function would also have range from $-inf$ to inf, so $g(z) = z$ or $g(z) = z^3$ would be good.

b) For $sin(x)$, a good activation function would be tanh, a scaled sigmoid. $sin(x)$ has range $[-1, 1]$ so a good activation function would have a similar range. $g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ which has range $(-1, 1)$

c) For the estimates of selling price of a house, a good activation function would be the relu. That is because the selling price will never be negative, and can reach a very high price. Thus, relu $g(z) = max(0, z)$ is the most fitting because it has range $[0, inf)$

d) For the probability that a currency image is a counterfeit, a good activation function would be the sigmoid, because the probability ranges from $[0, 1]$ and similarly, the sigmoid $g(z) = \frac{1}{1+e^{-z}}$ it will have range anywhere from $(0, 1)$

e) For the audio recording as input and classifying the sound as instruments, we would use softmax. This problem is a multiclass classification problem, because it will sort the sound into different classes (instruments). Softmax $g(x) = \frac{e^x}{\Sigma e^j}$ is good for a vector of predictions because it normalizes the data, and it is perfect for this problem to classify the instruments.

f) For deciding what major a student is based on their courses, a good activation function would be the sigmoid. The major that a student is in could be decided based a probability from $[0, 1]$ for each major. The sigmoid $g(z) = \frac{1}{1+e^{-z}}$ has range anywhere from $(0, 1)$, and will serve as a good final layer in the classification problem. An alternative would be the softmax $g(x) = \frac{e^x}{\Sigma e^j}$, because it can classify the major based on the courses, and it is a good activation function for a multiclass classification problem.

3

The dimensions are

$W^{[1]}$ $1x3$
$W^{[2]}$ $2x1$
$W^{[3]}$ $1x2$
$b^{[1]}$ $1x1$
$b^{[2]}$ $2x1$
$b^{[3]}$ $1x1$
$Z^{[1]}$ $1x50$
$Z^{[2]}$ $2x50$
$Z^{[3]}$ $1x50$
$A^{[1]}$ $1x50$
$A^{[2]}$ $2x50$
$A^{[3]}$ $1x50$

$A^{[3]}$ represents the output of each of the batch. Each of the element in the column of the matrix represents a single output the neural network of **one** batch.