Machine Learning Project 3 Neural Networks

Yifeng Qin, Pedram Safaei

November 8th

1 calculate_loss(model, X, y)

To calculate the loss we need to find the y hat to give to the softmax and cross entropy functions. We created a helper function (calculate_helper_variables(model,X, h_flag = False)) that calculates the helper variables described in the project description.

```
def calculate_helper_variables(model,X, h_flag = False):
  a = np.dot(X, model["W1"]) + model["b1"]
  h = np.tanh(a)
  z = np.dot(h, model["W2"]) + model["b2"]
  if h_flag:
      return z,h
  return z

```

This helper function will return the z helper variable and h if the h_flag is set true. We first calculate the helper variable a. It is found by taking X(training data) and dotting it with W1 and b1 which are the weights and bias. To calculate h, we take the helper a variable and pass it to the tanh which is our activation function to calculate the output. To calculate the z helper variable we take the calculated h value and dot it with the W2 added to b2. In calculate_loss(model, X, y) we set the helper variable z by calling the helper function. Then we calculate the y hat where the z value is passed to the softmax function to give us the prediction. The y hat and y label will be passed to cross entropy loss function. There it will sum over our training examples and add to the loss if the prediction is incorrect. The function will return the loss by multiplying it by -1 over the number of samples.

```
def calculate_loss(model, X, y):
  Loss = 0
  radius = y.shape[0]
  z = calculate_helper_variables(model,X)
  y_hat = np.exp(z) / (np.sum(np.exp(z), axis=1)).reshape(-1, 1)
  for i in range(radius):
      Loss = Loss + np.log(y_hat[i][y[i]])
  return (-1/radius) * Loss

```

2 predict(model, x)

The prediction function is very similar to the loss function. Instead of taking in the whole training data, it will be passed one sample without the label. To calculate the y hat we will call the helper function (calculate_helper_variables(model,X, h_flag = False) to give us our z helper variable. The softmax function will find the ratio of the exponential of the z value and the sum of exponential z values. All the probabilities will be between 0 and 1 and the sum of them all will come out to 1. The function will then append all the predictions into a list and return that list.

```
def predict(model, x):
  Prediction = []
  z = calculate_helper_variables(model,x)
```

```
    y_hat = np.exp(z - np.max(z)) / (np.exp(z - np.max(z))).sum()
    for i in y_hat:
        Prediction.append(np.argmax(i))
    return np.array(Prediction)

```

3 build_model(X, y, nn_hdim, num_passes=20000, print_loss=False)

This function is where all the training happens. For every epoch, which is the num_passes in this case we will predict all the features, run gradient decent and propegation and output the loss if necessary. According to the project description model is supposed to be a dictionary hence all weights and biases are saved in the model dictionary and it will get populated at the beginning of the build model using the initialize function (helper function so it was not included, refer to the program for more info)
Then we will run through a for loop that runs for the number of passes that were entered. First we will get our h and z helper variables from the helper function. We will then use back propagation to calculate the gradients:

$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$

$\frac{\partial L}{\partial a} = (1 - tanh^2 a) \bigcirc \frac{\partial L}{\partial \hat{y}} W_2^T$

$\frac{\partial L}{\partial W_2} = h^T \frac{\partial L}{\partial \hat{y}}$

$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}}$

$\frac{\partial L}{\partial W_1} = x^T \frac{\partial L}{\partial a}$

$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial a}$

Then we will update the weights and bias with these equations:

W1 = W1 - eta * $\frac{\partial L}{\partial W_1}$

W2 = W2 - eta * $\frac{\partial L}{\partial W_2}$

b1 = b1 - eta * $\frac{\partial L}{\partial b_1}$

b2 = b2 - eta * $\frac{\partial L}{\partial b_2}$

that is also included in a different update function in our program.

At the end If the print variable is true then we will print every 1000 iterations. At the end the function will return the model with the updated weights and biases.

```
W1,b1,W2,b2 = randomInitializer(y, nn_hdim)
model = {"W1" : W1, "W2": W2, "b1":b1, "b2":b2}
ylabel = _one_hot_values(y)
if (print_loss):
  print("New HiddenLayerSize")
for index in range(1,num_passes):
  z,h = calculate_helper_variables(model,X,True)
  y_hat = np.exp(z) / (np.sum(np.exp(z), axis=1, keepdims=True))
  dLdb1,dLdW1,dLdb2,dLdW2 = CalculateGradient(model , y_hat, ylabel, h)
  updateModel(model, dLdb1, dLdb2, dLdW1, dLdW2)

if (print_loss) and  not (index % 1000):
  print("iteration:", index, "Loss:", calculate_loss(model, X, y))
```

```
14 return model
15
```

You will notice that we are using one hot encoding, in general One hot encoding is used on categorical variables to convert them into a form that could be provided to ML algorithms to do a better job in prediction, here we found out that it is not just for doing a better job but it is also making it easier for us to pass in our labels for updating the model hence it will save a lot of time and make the training a lot faster. This function was taken from https://stackoverflow.com/questions/37292872/how-can-i-one-hot-encode-in-python and has been properly documented on the program, as this was not the purpose of the project we saw it fit to use this code or include it from the library, we decided to go with the first part because including the library one hot encoding would require us to play with the formatting and again this was just a helper function and not part of the project so we decided not to go on that route.

4 Plots