PROBLEMS ENCOUNTERED AND HOW TO SOLVE

莊晴雯/108062281

At the beginning of the task where we are asked to initialize parameters with He initialization, I was a little bit confused on what is the purpose of using random initialization and zero initialization for weight matrices and biases respectively. After doing some research, I found out that randomness is used to find a good enough set of weights for the specific mapping function. Moving forward, I encountered another issue when trying to implement the Sigmoid function. In the original function, there exists a condition such that when Z >= 0, then Sigmoid(Z) = (1/(1+exp(z))), otherwise, Sigmoid(Z) = exp(Z)/(1+exp(Z)). When implementing this to my code, I did

```
if Z >= 0:
    A = 1/(1 + np.exp(-Z))
    cache = Z
else:
    A = np.exp(Z)/(1 + np.exp(Z))
    cache = Z
```

```
tmp = []
for items in Z[0]:
  if items >= 0:
    tmp.append(1/(1 + np.exp(-items)))
    cache = Z
  else:
    tmp.append(np.exp(items)/(1 + np.exp(items)))
    cache = Z
A = np.array([tmp])
```

However, this yields an error since I just Z is not a number, but rather an array. Hence, I decided to iterate it.

Next, I also encountered a small problem when computing the Binary cross-entropy loss. I used np.sum and np.multiply to calculate the cost at first, and it computed the correct result but returns a wrong data type (numpy.float64 instead of numpy.ndarray). I figured that this must have something to do with the way I calculate the variable cost, and I noticed that I should use np.dot instead of np.multiply and np.sum to get numpy.ndarray as the data type.

BINARY CLASSIFIER IMPLEMENTATION

We are implementing the function L_layer_model to build the Binary Classifier. In this function, we first need to initialize the parameters by calling the function initialize_parameters_deep and pass layers_dims, which is the list containing the input size and each layer size as the parameter. We will get a dictionary with our parameters W_{1-l} and b_{1-l}.

Next, inside the for loop which iterates for a number of num_iterations times, we want to get the last post-activation value and a list of caches, and we can do so by calling the L_model_forward(X, parameters, classes) function. Afterwards, we can compute the Binary cross-entropy loss by calling the implemented compute_BCE_cost(AL, Y). Then, we can call L_model_backward(AL, Y, caches, classes) for the backward propagation which will return dictionary with gradients, which will be used when we update the parameters. Finally, we can call update_parameters(parameters, grads, learning_rate) to update the parameters using gradient descent on every W_l and b_l for l = 1, 2, …, L.

For the hyperparameter tuning, I used [4, 1] as the value of layer_dims, 0.0075 as the learning rate, and 3000 as the number of iterations. In the end, my accuracy value is 1.0.

```
layers_dims = [4, 1]
parameters = L_layer_model(X_train, y_train, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost = True, classes=2)
```

FUNCTION IMPLEMENTATIONS

To be able to implement the Binary Classifier, there are a few supporting functions that we need to implement beforehand.

initialize_parameters(n_x, n_y):

Calculate the parameters W1 and b1 using n_x and n_y, which are the size of input and output layer respectively. We use random initialization and multiply it by sqrt(2./n_x) for W1 and zero initialization and multiply it with sqrt(2./n_y) for b1.

initialize_parameters_deep(layer_dims):

Given the array with the dimensions of each layer as the values, we can use this to compute the parameters $W_{1\text{-}l}$ and $b_{1\text{-}l}$ using random and zero initialization and multiply it by sqrt(2./layer_dims[l-1]) where layer_dims[l-1] represents the dimension of the previous layer, since dimension of current layer is represented by layer_dims[l].

linear_forward(A, W, b):

Here, we just need to calculate Z, the pre-activation parameter, which is the dot product of W and A and then summed with b.

sigmoid(Z):

Calculates the activation value using the given formula: $\text{Sigmoid: } \sigma(Z) = \begin{cases} \frac{1}{1+e^{-Z}}, & \text{if } Z >= 0 \\ \frac{e^{Z}}{1+e^{Z}}, & \text{otherwise} \end{cases}$

relu(Z):

Relu(Z) returns the greater value between Z and 0.

linear_activation_forward(A_prev, W, b, activation):

In this function, we just need to call linear_forward(A_prev, W, b) and store it to Z and linear_cache, and depending on the value stored in the string activation, we call the respective function, either sigmoid or relu, and pass Z as the parameter.

compute_BCE_cost(AL, Y):

Calculate the cost using the given formula:

$$-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right))$$

linear_backward(dZ, cache):

Calculate dW, db, and dA_prev using the given formulas:

$$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

sigmoid_backward(dA, cache):

Calculate dZ using the formula $dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$, where $g'(Z^{[l]})$ can be found by counting the derivative of the sigmoid function using the formula: $\sigma'(Z^{[l]}) = \sigma(Z^{[l]})(1-\sigma(Z^{[l]}))$. This can be implied in Python code by:

```
dZ = dA * ((1/(1+np.exp(-Z))) * (1-(1/(1+np.exp(-Z)))))
```

relu_backward(dA, cache):

First convert dA to numpy.array and set dZ to 0 when Z <= 0.

linear_activation_backward(dA, cache, activation):

Depends on the value of the string activation (either “relu” or “sigmoid”), call relu_backward or sigmoid_backward to find the value of dZ which we will use to find the value of dA_prev, dW, and db by calling linear_backward(dZ, linear_cache).

ipdate_parameters(parameters, grads, learning_rate):

Here we just need to update the parameters Wl and bl for l = 1, 2, …, L using gradient descent: $W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}$, $b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$ where α is the learning rate. This can be done in Python using:

```
for l in range(L):
    parameters["W" + str(l+1)] = parameters['W' + str(l+1)] - learning_rate * grads['dW' + str(l+1)]
    parameters["b" + str(l+1)] = parameters['b' + str(l+1)] - learning_rate * grads['db' + str(l+1)]
```