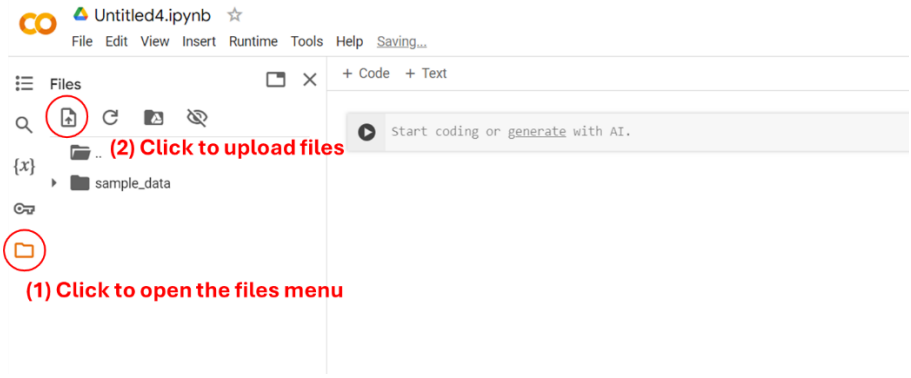
EE6310

Problem Set 11

Here, we will construct and train a neural network model through backpropagation. We will work with the logistic regression data set from the homework 8:

Metallic	**H%**	**Li%**	**Be%**	**…**	**U%**
y	x_1	x_2	x_3		x_{79}
1	0.41	0	0	…	0
0	0	0	0	…	0
0	0	0.04	0	…	0
…	…	…	…	…	

The full data can be downloaded from the weekly learning menu named "HW11.csv." To upload the csv file to Google Colab, Follow the following steps:



First you can read the data set with the following command:

```
import pandas as pd
df = pd.read_csv('HW11.csv')
data = df.to_numpy()
Y = data[:,0]
X = data[:,1:]
```

We will utilize the functions that we have wrote in the previous homework. Run these codes to save the functions

```
def layer_sizes(X, Y):
    """
    Arguments:
    X -- input dataset of shape (number of examples, input size)
    Y -- labels of shape (number of examples, output size)
```

```
    Returns:
    n_0 -- the size of the input layer
    n_1 -- the size of the hidden layer
    n_2 -- the size of the output layer
    """
    ### START CODE HERE ### (≈ 3 lines of code)
    n_0 = X.shape[1]
    n_1 = 4
    n_2 = 1
    ### END CODE HERE ###
    return (n_0, n_1, n_2)

def initialize_parameters(n_0, n_1, n_2):
    """
    Argument:
    n_0 -- size of the input layer
    n_1 -- size of the hidden layer
    n_2 -- size of the output layer

    Returns:
    params -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_0, n_1)
                    b1 -- bias vector of shape (n_1)
                    W2 -- weight matrix of shape (n_1, n_2)
                    b2 -- bias vector of shape (n_2)
    """

    np.random.seed(2) # we set up a seed so that your output matches
ours although the initialization is random.

    ### START CODE HERE ### (≈ 4 lines of code)
```

```
    W1 = np.random.randn(n_0,n_1)*0.01
    b1 = np.zeros((n_1,))
    W2 = np.random.randn(n_1,n_2)*0.01
    b2 = np.zeros((n_2,))
    ### END CODE HERE ###

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters

def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (m, n_0)
    parameters -- python dictionary containing your parameters
(output of initialization function)

    Returns:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    ### END CODE HERE ###
```

```
    # Implement Forward Propagation to calculate A2 (probabilities)
    ### START CODE HERE ### (≈ 4 lines of code)
    Z1 = X@W1 + b1
    A1 = np.tanh(Z1)
    Z2 = A1@W2 + b2
    A2 = 1/(1+np.exp(-Z2))
    ### END CODE HERE ###

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache
```

(a) Let's write the function to calculate the cost function by filling the code between "### START CODE HERE ###" and "### END CODE HERE ###"

Note that we have error in slides. The formula involving the calculation of cost function is:

$L(\boldsymbol{A}^{[2]}, \boldsymbol{y}) = -\boldsymbol{y}\log(\boldsymbol{A}^{[2]}) - (1-\boldsymbol{y})\log(1-\boldsymbol{A}^{[2]})$

$J(\boldsymbol{W}^{[1]}, \boldsymbol{b}^{[1]}, \boldsymbol{W}^{[2]}, \boldsymbol{b}^{[2]}) = \bar{L}$

```
def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation (13)

    Arguments:
    A2 -- The sigmoid output of the second activation, of shape
(number of examples, 1)
    Y -- "true" labels vector of shape (number of examples)
    parameters -- python dictionary containing your parameters W1,
b1, W2 and b2
```

```
    Returns:
    cost -- cross-entropy cost given equation (13)
    """
    # Remove the feature (the last) dimension of the A2
    ### START CODE HERE ### (≈ 1 line of code)



    ### END CODE HERE ###

    # Compute the cross-entropy cost
    ### START CODE HERE ### (≈ 2 lines of code)



    ### END CODE HERE ###

    return Jprint ('The shape of X is: ' + str(shape_X))

n_0, n_1, n_2 = layer_sizes(X, Y)
parameters = initialize_parameters(n_0, n_1, n_2)
A2, cache = forward_propagation(X, parameters)
J = compute_cost(A2, Y, parameters)
print("cost = " + str(J))
assert isinstance(J, float), 'Wrong answer!'
assert np.round(J,2) == 0.37
```

(b) Implement the backpropagation by filling the code between "### START CODE HERE ###" and "### END CODE HERE ###"

```
def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation using the instructions above.
```

```
    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (number of examples, n_0)
    Y -- "true" labels vector of shape (number of examples)

    Returns:
    grads -- python dictionary containing your gradients with
respect to different parameters
    """
    m = Y.shape[0]
    # Retrieve also A1 and A2 from dictionary "cache".
    ### START CODE HERE ### (≈ 2 lines of code)



    ### END CODE HERE ###

    # Add a dimension to Y
    ### START CODE HERE ### (≈ 1 lines of code)

    ### END CODE HERE ###

    # Backward propagation: calculate dW1, db1, dW2, db2.
    ### START CODE HERE ### (≈ 6 lines of code, corresponding to 6
equations on slide above)

    ### END CODE HERE ###

    grads = {"dJdW1": dJdW1,
             "dJdb1": dJdb1,
             "dJdW2": dJdW2,
```

```
             "dJdb2": dJdb2}

    return grads

grads = backward_propagation(parameters, cache, X, Y)
print ("dJdW1 = "+ str(grads["dJdW1"]))
print ("dJdb1 = "+ str(grads["dJdb1"]))
print ("dJdW2 = "+ str(grads["dJdW2"]))
print ("dJdb2 = "+ str(grads["dJdb2"]))
assert grads["dJdW2"].shape == (4,1), 'Wrong answer!'
assert grads["dJdb2"].shape == (1,), 'Wrong answer!'
assert grads["dJdW1"].shape == (79,4), 'Wrong answer!'
assert grads["dJdb1"].shape == (4,), 'Wrong answer!'
assert np.round(np.mean(grads["dJdW1"]),5) == -0.00039, 'Wrong
answer!'
```

(c) Now let's use the calculated partial derivative to update the parameters:

```
def update_parameters(parameters, grads, learning_rate = 0.1):
    """
    Updates parameters using the gradient descent update rule given
above

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients

    Returns:
    parameters -- python dictionary containing your updated
parameters
    """
    # Retrieve each parameter from the dictionary "parameters"
```

```
    ### START CODE HERE ### (≈ 4 lines of code)




    ### END CODE HERE ###

    # Retrieve each gradient from the dictionary "grads"
    ### START CODE HERE ### (≈ 4 lines of code)




    ## END CODE HERE ###

    # Update rule for each parameter
    ### START CODE HERE ### (≈ 4 lines of code)




    ### END CODE HERE ###

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters

parameters = update_parameters(parameters, grads)
```

```
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

(d) Finally, let's write the function to train out model.

```
def nn_model(X, Y, n_1, num_iterations = 100000, print_cost=False):
    """
    Arguments:
    X -- dataset of shape (number of examples, n_0)
    Y -- labels of shape (number of examples)
    n_h -- size of the hidden layer
    num_iterations -- Number of iterations in gradient descent loop
    print_cost -- if True, print the cost every 10000 iterations

    Returns:
    parameters -- parameters learnt by the model. They can then be
used to predict.
    """

    np.random.seed(3)
    n_0, _, n_2 = layer_sizes(X, Y)

    # Initialize parameters. Inputs: "n_0, n_1, n_2"
    ### START CODE HERE ### (≈ 1 lines of code)

    ### END CODE HERE ###

    # Loop (gradient descent)
```

```
    for i in range(0, num_iterations):

        ### START CODE HERE ### (≈ 4 lines of code)
        # Forward propagation. Inputs: "X, parameters". Outputs:
"A2, cache".



        # Cost function. Inputs: "A2, Y, parameters". Outputs:
"cost".

        # Backpropagation. Inputs: "parameters, cache, X, Y".
Outputs: "grads".

        # Gradient descent parameter update. Inputs: "parameters,
grads". Outputs: "parameters".

        ### END CODE HERE ###

        # Print the cost every 1000 iterations
        if print_cost and i % 10000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return parameters

parameters = nn_model(X, Y, 4, num_iterations=100000,
print_cost=True)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

(e) Let's write the code to make predictions using the trained model

```
def predict(parameters, X):
    """
    Using the learned parameters, predicts a class for each example
in X

    Arguments:
    parameters -- python dictionary containing your parameters
    X -- input data of size (m, n_0)

    Returns
    y_hat -- vector of predictions of our model
    """

    # Computes probabilities using forward propagation, and
classifies to 0/1 using 0.5 as the threshold.
    ### START CODE HERE ### (≈ 2 lines of code)



    ### END CODE HERE ###

    # Remove the last dimension of the y_hat
    ### START CODE HERE ### (≈ 1 lines of code)

    ### END CODE HERE ###

    return y_hat
```

```
y_hat = predict(parameters, X)
print("predictions mean = " + str(np.mean(y_hat)))

assert np.round(np.mean(y_hat),3) == 0.545, 'Wrong answer!'
"A2": A2}

    return A2, cache
```

(f) Finally let's write the code to calculate accuracy. Google about how to calculate the accuracy for binary classification.

```
def caclulate_accuracy(y_hat, Y):
    """
    Using the y_hat and Y, calculate the accuracy

    Arguments:
    y_hat -- predicted Y values (m,)
    Y -- actual Y values (m, )

    Returns
    accuracy -- a float value of accuracy
    """

    #
    ### START CODE HERE ### (≈ 2 lines of code)
    accuracy = np.mean(Y==y_hat)
    ### END CODE HERE ###

    return accuracy

accuracy = caclulate_accuracy(y_hat, Y)
```

```
print("Accuracy = " + str(np.mean(y_hat)))
assert np.round(np.mean(accuracy),3) == 0.837, 'Wrong answer!'
```