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  y | H%  *x1* | Li%  *x2* | Be%  *x3* | … | U%  *x79* |
| 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:

A screenshot of a computer

Description automatically generated

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 |

1. 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:

|  |
| --- |
| 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 |

1. 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!' |

1. 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"])) |

1. 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"])) |

1. 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 |

1. 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!' |