Deep_Learning_Lab_Exam_Unsolved_Fall_2021

November 8, 2024

1 Deep Learning Lab Exam

In this lab exam, you will be implementing a neural network with 1 hidden layer (with the activation function tanh) and use it to classify whether a banknote is authentic (**1**) or not (**0**): Your network would look like the following:

2 Exam set up (Do not edit!)

[1]:
```
# Download the dataset using wget. If you are not on colab or linux, you can␣
↪skip this line and download your dataset directly
# and place it in the same folder as your notebook
!wget http://archive.ics.uci.edu/ml/machine-learning-databases/00267/
↪data_banknote_authentication.txt
```

```
--2024-11-08 09:55:15--  http://archive.ics.uci.edu/ml/machine-learning-
databases/00267/data_banknote_authentication.txt
Resolving archive.ics.uci.edu (archive.ics.uci.edu)…

128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: unspecified
Saving to: ‘data_banknote_authentication.txt’

data_banknote_authe     [   <=>               ]  45.31K  76.2KB/s    in 0.6s

2024-11-08 09:55:16 (76.2 KB/s) - ‘data_banknote_authentication.txt’ saved
[46400]
```

[25]:
```
# Import relevant libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import pandas as pd
```

Dataset Attribute Information:

1. variance of Wavelet Transformed image (continuous)
2. skewness of Wavelet Transformed image (continuous)
3. curtosis of Wavelet Transformed image (continuous)
4. entropy of image (continuous)
5. class (integer)

[26]:
```
# Importing the dataset and extracting the inputs and the outputs
data = np.genfromtxt("data_banknote_authentication.txt", delimiter = ",")
X = data[:,:4]
y = data[:, 4]
```

[27]:
```
# Split the dataset into a training and test dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
X_train = X_train.T
y_train = y_train.reshape(1, y_train.shape[0])
X_test = X_test.T
y_test = y_test.reshape(1, y_test.shape[0])
print ("Train X Shape: ", X_train.shape)
print ("Train Y Shape: ", y_train.shape)
print ("%d training examples." % (X_train.shape[1]))

print ('\nTest X Shape: ', X_test.shape)
```

```
Train X Shape:  (4, 1097)
Train Y Shape:  (1, 1097)
1097 training examples.

Test X Shape:  (4, 275)
```

Note that in X, every column contains each record

3 The exam starts

In this exam, we will: 1. Define the neural network structure (# of input neurons, # of hidden neurons, etc). 2. Initialize the model's parameters 3. Loop: - Implement forward propagation - Compute loss - Implement backward propagation to get the gradients - Update parameters (gradient descent) 4. Evaluate accuracy

3.0.1 [1 mark] Define Structure:

In this task, you need to take the inputs (**X**) and output(**Y**) and number of hidden neurons you want to add in your hidden layer, and based on that return number of input neurons, number of hidden neurons and number of output neurons.

[28]:
```python
def define_structure(X, Y,hidden_neurons):
  # IMPLEMENT
  num_input_neurons = 4
  num_hidden_neurons = hidden_neurons
  num_output_neurons = 1
  return num_input_neurons, num_hidden_neurons, num_output_neurons

num_input_neurons, num_hidden_neurons, num_output_neurons =␣
 ↪define_structure(X_train, y_train,4)
print("The size of the input layer is := " + str(num_input_neurons))
print("The size of the hidden layer is := " + str(num_hidden_neurons))
print("The size of the output layer is := " + str(num_output_neurons))

# Note that input layer should always have 4 neurons since we have 1 input
# Output layer should have 1 neuron since we are predicting 1 value, whether
# the note is authentic or not

# For the purpose of this exam, number of hidden neurons would be 4.
```

```
The size of the input layer is := 4
The size of the hidden layer is := 4
The size of the output layer is := 1
```

3.0.2 [1 mark] : Parameter Initialization

Now that we know the structure of the network, we can initialize the parameters of our network.

Since our network has 1 hidden layer, we will have 2 set of weights and biases. 1. From input to hidden layer (W1, b1) 2. from Hidden to output layer (W2, b2)

Based on the structure of the network, initialize these parameters.

Initialize weights (W1,W2) using `np.random.randn` and scale them by a factor of 0.01.

Initialize biases (b1,b2) as 0s (using `np.zeros`)

[29]:
```python
def parameters_initialization(num_input_neurons, num_hidden_neurons,␣
 ↪num_output_neurons):
  # Seed is set for ease of testing. Do not remove.
  np.random.seed(2)
  # IMPLEMENT

  W1 = np.random.randn(num_hidden_neurons, num_input_neurons) * 0.01
  b1 = np.zeros((num_hidden_neurons, 1))

  W2 = np.random.randn(num_output_neurons, num_hidden_neurons) * 0.01
  b2 = np.zeros((num_output_neurons, 1))


  # We are going to be storing the parameters in a
```

```python
    # dictionary for ease of access later on
    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

3.0.3 [0.5 marks]: Implement the Sigmoid function

Sigmoid is defined as follows:

$$S(z) = \frac{1}{1 + e^{-z}}$$

[30]:
```python
def sigmoid(z):
    # IMPLEMENT
    denom = 1 + np.exp(-z)
    sig = np.divide(1, denom)
    return sig
```

3.0.4 [1 mark] Forward propogation

In order to get a result from the neural network, we need to propogate our inputs through the network.

Your goal is to implement a function which propogates the inputs (X) through the network and computes the result at each part of the network.

We are naming the parts of the networks as follows:

Z_1 as the output to the hidden layer from the input layer (without the activation function applied)

A_1 as applying the activation function to Z_1

Z_2 as the output to the output layer from the hidden layer (A_1) with no activation function

A_2 as sigmoid being applied to Z_2 to get the final output (Probability that the input belongs to class 1 by the network)

$$Z_1 = W_1 X + b_1$$
$$A_1 = \tanh(Z_1)$$
$$Z_2 = W_2 A_1 + b_2$$
$$A_2 = S(Z_2)$$

Note: Make sure your matrices are in such a way that your weight matrices are initialized in such a way, such that these equations make sense. If you don't, then you might have to adjust how you deal with backpropogation.

In numpy, you can do tanh, by `np.tanh`

[31]:
```
def forward_propagation(X, parameters):
  # Extract weights and biases from the dictionary
  W1 = parameters['W1']
  b1 = parameters['b1']
  W2 = parameters['W2']
  b2 = parameters['b2']

  # Compute Z1,A1,Z2,A2
  # IMPLEMENT

  Z1 = np.dot(W1, X) + b1
  A1 = np.tanh(Z1)
  Z2 = np.dot(W2, A1) + b2
  A2 = sigmoid(Z2)


  # Save the output of each layer in a dictionary to help with backpropogation␣
↪later on
  cache = {"Z1": Z1,"A1": A1,"Z2": Z2,"A2": A2}
  # Return the final output and output at each layer
  return A2, cache
```

3.0.5 [1.5 marks]: Binary Cross-entropy loss

In order for you to evaluate how far your model is from correct predictions, you need to have a loss function. We will be using Binary Cross-entropy loss since our prediction can be either Class 0 or Class 1. The formula to compute Binary Cross-entropy loss J using actual outputs Y and the **output from your last layer** $A^{[2]}$ is as follows

$$J(Y, A^{[2]}) = -\frac{1}{m}\sum_{i=1}^{m} Y^{(i)}\log(A^{[2](i)}) + (1 - Y^{(i)})\log(1 - A^{[2](i)})$$

where m is the number of samples.

You can use `np.log` for log.

[36]:
```
def cross_entropy_cost(Y,A2):
  # IMPLEMENT
  p1 = np.multiply(Y, np.log(A2)); p2 = np.multiply(1 - Y, np.log(1 - A2))
  summ = np.sum(p1 + p2)
  cost = np.multiply(np.divide(-1, Y.shape[1]), summ)
  return cost
```

3.0.6 [1.5 marks] : Back Propogation

Now that we have gotten the output at each layer, we can compute the gradients at each layer and use that in order to nudge our model parameters in the right direction.

To figure out how much our actual output deviates from our desired output.

$$dZ^{[2]} = A^{[2]} - Y$$

and using that we can compute the changes for the weights and biases in the hidden layer

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$$

$db^{[2]}$ is what you can get if you sum each row for $dZ^{[2]}$ and then divide this sum by number of samples m (Do keep dimensions in numpy!)

To compute the changes in the input layer weights

$$dZ^{[1]} = W^{[2]^T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$$

Where $g^{[1]}$ is your activation function (in our case is tanh)

The derivative of tanh is

$$\tanh(Z) = 1 - \tanh^2(Z)$$

Note that $*$ means element-wise multiplication. In numpy, for matrix multiplication use np.dot, and for elementwise you can use `*`.

Using that, we can compute the change needed in weights and biasses.

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

$db^{[1]}$ is what you can get if you sum each row for $dZ^{[1]}$ and then divide this sum by number of samples m (Do keep dimensions in numpy!)

Note: sum each row for means that if I have an array `[[1,2],[3,5]]`, we will get `[3,8]` and keeping the dimensions same, would give us `[[3],[8]]`.

To raise each element in a numpy array A to the power i, you can can be done by `np.power(A,i)`

[37]:
```
def backward_propagation(parameters, cache, X, Y):

    W1 = parameters['W1']
    W2 = parameters['W2']
    A1 = cache['A1']
    A2 = cache['A2']

    # Compute dW2,db2,dW1,db1
    # You would find it convenient it to compute dZ2,dZ1
    # IMPLEMENT
    Z1 = cache['Z1']
    dZ2 = A2 - Y
    dW2 = np.multiply(np.divide(1, Y.shape[1]), np.dot(dZ2, A1.T))
    dZ1 = np.multiply(np.dot(W2.T, dZ2), (1 - np.tanh(Z1)**2))
    dW1 = np.multiply(np.divide(1, Y.shape[1]), np.dot(dZ1, X.T))
```

```
    db2 = np.multiply(np.divide(1, Y.shape[1]), np.sum(dZ2, axis=1,␣
↪keepdims=True))
    db1 = np.multiply(np.divide(1, Y.shape[1]), np.sum(dZ1, axis=1,␣
↪keepdims=True))

    # Store the changes in a dictionary grads to be used for gradient descent
    grads = {"dW1": dW1, "db1": db1, "dW2": dW2,"db2": db2}

    return grads
```

3.0.7 [1.5 marks]: Gradient Descent

Now that we have computed the changes that we need, we can just apply gradient descent to every weight and bias parameter with learning rate α

$W = W - \alpha dW$ and $B = b - \alpha db$

[38]:
```
def gradient_descent(parameters, grads, learning_rate = 0.01):
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']

    # Perform gradient descent
    # IMPLEMENT

    W1 = W1 - np.multiply(learning_rate, dW1)
    b1 = b1 - np.multiply(learning_rate, db1)
    W2 = W2 - np.multiply(learning_rate, dW2)
    b2 = b2 - np.multiply(learning_rate, db2)

    # Return the updated parameters
    parameters = {"W1": W1, "b1": b1,"W2": W2,"b2": b2}

    return parameters
```

3.0.8 [1.5 marks] Making the network: Bringing it all together.

Now we can bring everything together and make our model work.

As a reminder, the network is supposed to

1. Define the neural network structure (# of input neurons, # of hidden neurons, etc).

2. Initialize the model's parameters
3. Loop:
    - Implement forward propagation
    - Compute loss (print the loss)
    - Implement backward propagation to get the gradients
    - Update parameters (gradient descent)

Once the model is trained, we can return the parameters.

[39]:
```
def neural_network_model(X, Y, num_hidden_neurons, num_iterations = 1000):
    np.random.seed(3)
    # Define the structure
    # Initialize Parameters
    # IMPLEMENT

    num_input_neurons, num_hidden_neurons, num_output_neurons = 
 ↪define_structure(X, Y,num_hidden_neurons)
    parameters = parameters_initialization(num_input_neurons, 
 ↪num_hidden_neurons, num_output_neurons)

    # Training loop

    for i in range(num_iterations):
      # forward prop
      A2, cache = forward_propagation(X, parameters)
      # Compute loss
      loss = cross_entropy_cost(Y,A2)
      # Back prop to find gradients
      grads = backward_propagation(parameters, cache, X, Y)
      # Update gradients
      parameters = gradient_descent(parameters, grads)
      # IMPLEMENT
      # Print loss (Do not edit!)
      if i % 5 == 0:
          print ("Cost after iteration %i: %f" %(i, loss))
    return parameters

# Train the model (Do not edit!!)
parameters = neural_network_model(X_train, y_train, 4, num_iterations=1000)
```

```
Cost after iteration 0: 0.692975
Cost after iteration 5: 0.692665
Cost after iteration 10: 0.692351
Cost after iteration 15: 0.692028
Cost after iteration 20: 0.691692
Cost after iteration 25: 0.691338
Cost after iteration 30: 0.690962
Cost after iteration 35: 0.690557
```

```
Cost after iteration 40: 0.690117
Cost after iteration 45: 0.689636
Cost after iteration 50: 0.689107
Cost after iteration 55: 0.688521
Cost after iteration 60: 0.687873
Cost after iteration 65: 0.687152
Cost after iteration 70: 0.686352
Cost after iteration 75: 0.685463
Cost after iteration 80: 0.684477
Cost after iteration 85: 0.683388
Cost after iteration 90: 0.682187
Cost after iteration 95: 0.680866
Cost after iteration 100: 0.679420
Cost after iteration 105: 0.677843
Cost after iteration 110: 0.676127
Cost after iteration 115: 0.674268
Cost after iteration 120: 0.672259
Cost after iteration 125: 0.670096
Cost after iteration 130: 0.667773
Cost after iteration 135: 0.665286
Cost after iteration 140: 0.662629
Cost after iteration 145: 0.659797
Cost after iteration 150: 0.656787
Cost after iteration 155: 0.653594
Cost after iteration 160: 0.650215
Cost after iteration 165: 0.646648
Cost after iteration 170: 0.642889
Cost after iteration 175: 0.638938
Cost after iteration 180: 0.634794
Cost after iteration 185: 0.630457
Cost after iteration 190: 0.625929
Cost after iteration 195: 0.621210
Cost after iteration 200: 0.616305
Cost after iteration 205: 0.611217
Cost after iteration 210: 0.605950
Cost after iteration 215: 0.600512
Cost after iteration 220: 0.594907
Cost after iteration 225: 0.589145
Cost after iteration 230: 0.583233
Cost after iteration 235: 0.577180
Cost after iteration 240: 0.570995
Cost after iteration 245: 0.564687
Cost after iteration 250: 0.558268
Cost after iteration 255: 0.551746
Cost after iteration 260: 0.545132
Cost after iteration 265: 0.538436
Cost after iteration 270: 0.531668
Cost after iteration 275: 0.524840
```

```
Cost after iteration 280: 0.517962
Cost after iteration 285: 0.511044
Cost after iteration 290: 0.504098
Cost after iteration 295: 0.497134
Cost after iteration 300: 0.490163
Cost after iteration 305: 0.483197
Cost after iteration 310: 0.476244
Cost after iteration 315: 0.469317
Cost after iteration 320: 0.462425
Cost after iteration 325: 0.455576
Cost after iteration 330: 0.448781
Cost after iteration 335: 0.442047
Cost after iteration 340: 0.435381
Cost after iteration 345: 0.428791
Cost after iteration 350: 0.422283
Cost after iteration 355: 0.415862
Cost after iteration 360: 0.409532
Cost after iteration 365: 0.403299
Cost after iteration 370: 0.397165
Cost after iteration 375: 0.391133
Cost after iteration 380: 0.385205
Cost after iteration 385: 0.379383
Cost after iteration 390: 0.373667
Cost after iteration 395: 0.368060
Cost after iteration 400: 0.362561
Cost after iteration 405: 0.357170
Cost after iteration 410: 0.351887
Cost after iteration 415: 0.346712
Cost after iteration 420: 0.341643
Cost after iteration 425: 0.336679
Cost after iteration 430: 0.331820
Cost after iteration 435: 0.327064
Cost after iteration 440: 0.322409
Cost after iteration 445: 0.317854
Cost after iteration 450: 0.313397
Cost after iteration 455: 0.309037
Cost after iteration 460: 0.304771
Cost after iteration 465: 0.300598
Cost after iteration 470: 0.296516
Cost after iteration 475: 0.292523
Cost after iteration 480: 0.288616
Cost after iteration 485: 0.284795
Cost after iteration 490: 0.281056
Cost after iteration 495: 0.277399
Cost after iteration 500: 0.273820
Cost after iteration 505: 0.270319
Cost after iteration 510: 0.266893
Cost after iteration 515: 0.263540
```

```
Cost after iteration 520: 0.260259
Cost after iteration 525: 0.257048
Cost after iteration 530: 0.253905
Cost after iteration 535: 0.250828
Cost after iteration 540: 0.247815
Cost after iteration 545: 0.244866
Cost after iteration 550: 0.241977
Cost after iteration 555: 0.239148
Cost after iteration 560: 0.236378
Cost after iteration 565: 0.233663
Cost after iteration 570: 0.231004
Cost after iteration 575: 0.228398
Cost after iteration 580: 0.225845
Cost after iteration 585: 0.223343
Cost after iteration 590: 0.220889
Cost after iteration 595: 0.218485
Cost after iteration 600: 0.216127
Cost after iteration 605: 0.213815
Cost after iteration 610: 0.211547
Cost after iteration 615: 0.209323
Cost after iteration 620: 0.207141
Cost after iteration 625: 0.205000
Cost after iteration 630: 0.202900
Cost after iteration 635: 0.200838
Cost after iteration 640: 0.198815
Cost after iteration 645: 0.196829
Cost after iteration 650: 0.194879
Cost after iteration 655: 0.192964
Cost after iteration 660: 0.191084
Cost after iteration 665: 0.189237
Cost after iteration 670: 0.187423
Cost after iteration 675: 0.185641
Cost after iteration 680: 0.183890
Cost after iteration 685: 0.182170
Cost after iteration 690: 0.180479
Cost after iteration 695: 0.178817
Cost after iteration 700: 0.177183
Cost after iteration 705: 0.175577
Cost after iteration 710: 0.173998
Cost after iteration 715: 0.172445
Cost after iteration 720: 0.170918
Cost after iteration 725: 0.169415
Cost after iteration 730: 0.167938
Cost after iteration 735: 0.166484
Cost after iteration 740: 0.165054
Cost after iteration 745: 0.163646
Cost after iteration 750: 0.162261
Cost after iteration 755: 0.160898
```

```
Cost after iteration 760: 0.159556
Cost after iteration 765: 0.158234
Cost after iteration 770: 0.156934
Cost after iteration 775: 0.155653
Cost after iteration 780: 0.154392
Cost after iteration 785: 0.153150
Cost after iteration 790: 0.151926
Cost after iteration 795: 0.150721
Cost after iteration 800: 0.149534
Cost after iteration 805: 0.148364
Cost after iteration 810: 0.147212
Cost after iteration 815: 0.146076
Cost after iteration 820: 0.144957
Cost after iteration 825: 0.143854
Cost after iteration 830: 0.142767
Cost after iteration 835: 0.141695
Cost after iteration 840: 0.140639
Cost after iteration 845: 0.139597
Cost after iteration 850: 0.138570
Cost after iteration 855: 0.137557
Cost after iteration 860: 0.136558
Cost after iteration 865: 0.135573
Cost after iteration 870: 0.134602
Cost after iteration 875: 0.133643
Cost after iteration 880: 0.132698
Cost after iteration 885: 0.131765
Cost after iteration 890: 0.130845
Cost after iteration 895: 0.129937
Cost after iteration 900: 0.129041
Cost after iteration 905: 0.128156
Cost after iteration 910: 0.127284
Cost after iteration 915: 0.126422
Cost after iteration 920: 0.125572
Cost after iteration 925: 0.124733
Cost after iteration 930: 0.123904
Cost after iteration 935: 0.123086
Cost after iteration 940: 0.122279
Cost after iteration 945: 0.121481
Cost after iteration 950: 0.120694
Cost after iteration 955: 0.119916
Cost after iteration 960: 0.119148
Cost after iteration 965: 0.118390
Cost after iteration 970: 0.117641
Cost after iteration 975: 0.116901
Cost after iteration 980: 0.116170
Cost after iteration 985: 0.115447
Cost after iteration 990: 0.114734
Cost after iteration 995: 0.114029
```

[ ]:

```

```

3.0.9 [0.5 marks] Prediction and accuracy:

We now have a well trained model. We can now evaluate how good our model is on the test dataset, your goal is to create a function `prediction` which takes in parameters and the test inputs and based on that give predictions.

If we have probability of class 1 as > 0.5, our prediction is 1, else 0.

Once you are done with that compute the accuracy: See the **percentage** of predictions that were right (This will be done using actual Ys and your predictions) on both the **training and test** dataset

[44]:

```
def prediction(parameters, X):
    # IMPLEMENT
    A2, cache = forward_propagation(X, parameters)

    prediction = (A2 > 0.5).astype(int)
    return prediction

# Feel free to make a function for accuracy as well
```

[45]:

```
predictions = prediction(parameters, X_train)
# Feel free to make an accuracy function to compute accuracy
predictions = prediction(parameters, X_test)
```

[46]:

```
predictions
```

[46]:

```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0,
        1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0,
        1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
        0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1,
        1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1,
        1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0,
        0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0,
        1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,
        1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0,
        0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1,
        1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1,
        0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1,
        1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0]])
```