Diagram

Description automatically generated with medium confidence

**Homework 2: Backpropagation**

**Objectives**

* To implement the feedforward process on a hardcoded neural network architecture
* To understand and implement backpropagation for a specific neural network
* To implement the training procedure for a specific neural network

**Submitted by:**

|  |  |
| --- | --- |
| **Name** | **ID** |
| Youssef Ghoneim | B00087523 |
| Mohammed Fawaz | B00087431 |

**Submitted to:**

Dr. Imran Zualkernan

Mr. Ali Reza Sajun

**Submitted on: 12/10/2023**

**Refer to the Backpropagation Slides for this Homework.**

**Question 1: Hardcoding Forward and Backpropagation (4 marks)**

The objective of this question is to hardcode the forward, backpropagation, and train methods for a 2-3-1 neural network as below:

1. Create and use a dataset of x + y = z, where x and y are inputs and z is the output. Assume x and y are positive numbers.
2. Modify the class below to reflect the architecture of the 2-3-1 neural network shown above. Use sigmoid as the activation function for the final layer.
3. Split your data into an 80-20 split and run training for 100 epochs

Deliverables:

1. Paste the modified code representing the solution to training the 2-3-1 network in the word document. Make sure that the code is readable.
2. Print out the training results and comment on them. Paste a screenshot of your results and your comments in the solution document.

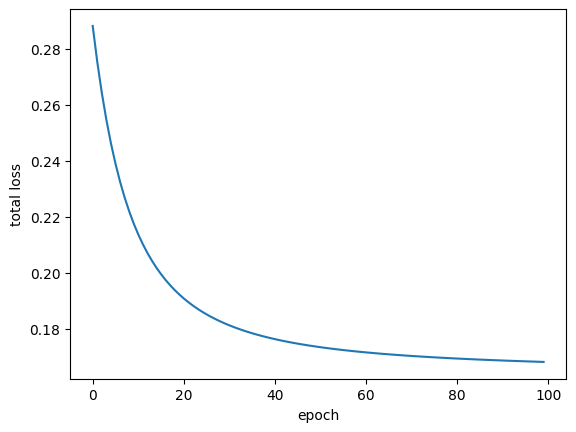
If needed, build upon the given skeleton code.

**Skeleton code:**

|  |
| --- |
| import numpy as np  class HardcodedNeuralNetwork:  def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):  self.input\_dim = input\_dim  self.hidden\_dim = hidden\_dim  self.output\_dim = output\_dim    # Initialize the parameters of the neural network  self.weights1 = np.random.randn(self.input\_dim, self.hidden\_dim)  self.bias1 = np.zeros((1, self.hidden\_dim))  self.weights2 = np.random.randn(self.hidden\_dim, self.output\_dim)  self.bias2 = np.zeros((1, self.output\_dim))    def sigmoid(self, x):  return 1 / (1 + np.exp(-x))  def relu(x):  return np.maximum(0, x)    def forward\_propagation(self, x):    return a1, output    def compute\_loss(self, x, y):  # Compute the loss (Mean Squared Error) of the neural network  \_, output = self.forward\_propagation(x)  num\_examples = len(x)  loss = np.mean(np.square(y - output))  return loss    def backward\_propagation(self, x, y, a1, output, learning\_rate):  # To perform backpropogation do the following steps  # 1. Calculate delta for last layer  # 2. Calculate delta for previous layer  # 3. Calculate DL/dw for all weights  # 4. Update the weights  # 5. Rerturn the weights    def train(self, x, y, num\_epochs, learning\_rate):  for epoch in range(num\_epochs):  print(f"Epoch {epoch + 1}: Loss = {loss}")    def predict(self, x):  \_, output = self.forward\_propagation(x)  predictions = np.argmax(output, axis=1)  return predictions |

Modified code provided below:

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  from sklearn.model\_selection import train\_test\_split  class HardcodedNeuralNetwork:  def \_\_init\_\_(self, input\_dim, hidden\_dim, output\_dim):  self.input\_dim = input\_dim  self.hidden\_dim = hidden\_dim  self.output\_dim = output\_dim  # Initialize the parameters of the neural network  self.weights1 = np.random.randn(self.hidden\_dim, self.input\_dim)  self.bias1 = np.zeros((self.hidden\_dim, 1))  self.weights2 = np.random.randn(self.output\_dim, self.hidden\_dim)  self.bias2 = np.zeros((self.output\_dim, 1))  def sigmoid(self, x):  return 1 / (1 + np.exp(-x))  def derivativeSigmoid(self, x):  return (self.sigmoid(x) \* (1 - self.sigmoid(x)))  def relu(x):  return np.maximum(0, x)  def forward\_propagation(self, x):  input = np.array(x)  z1 = np.dot(self.weights1, input) + self.bias1  a1 = self.sigmoid(z1)  z2 = np.dot(self.weights2, a1) + self.bias2  output = self.sigmoid(z2)  return z1, a1, z2, output  def compute\_loss(self, x, y):  # Compute the loss (Mean Squared Error) of the neural network  \_, \_, \_, output = self.forward\_propagation(x)  num\_examples = len(x)  loss = np.mean(np.square(y - output))  return loss  def backward\_propagation(self, x, y, a1, output, z1, z2, learning\_rate):  # To perform backpropogation do the following steps  # 1. Calculate delta for last layer  deltas2 = ((y-output)\*(self.derivativeSigmoid(z2)))  # 2. Calculate delta for previous layer  deltas1 = self.weights2.T.dot(deltas2)\*(self.derivativeSigmoid(z1))  # 3. Calculate DL/dw for all weights  # print('deltas1', deltas1.shape)  # print('deltas2', deltas2.shape)  # print('a1', a1.shape)  # print('a2', output.shape)  dw1 = np.dot(deltas1, x.T)  dw2 = np.dot(deltas2, a1.T)  # print('weights1', self.weights1.shape)  # print('weights2', self.weights2.shape)  # print('dw1',dw1.shape)  # print('dw2', dw2.shape)  # 4. Update the weights  updated\_weights1 = self.weights1 + learning\_rate \* dw1  updated\_weights2 = self.weights2 + learning\_rate \* dw2  # 5. Return the weights  return updated\_weights1, updated\_weights2  def train(self, x, y, num\_epochs, learning\_rate):  m = y.shape[1]  loss\_history = []  for epoch in range(num\_epochs):  epoch\_loss = 0  for i in range(m): # Iterate through the samples  x\_batch = x[:, i].reshape(2, 1)  y\_batch = y[:, i].reshape(1, 1)  z1, a1, z2, output = self.forward\_propagation(x\_batch)  loss = self.compute\_loss(x\_batch, y\_batch)  self.weights1, self.weights2 = self.backward\_propagation(x\_batch, y\_batch, a1, output, z1, z2, learning\_rate)  epoch\_loss = epoch\_loss + loss  avg\_loss = epoch\_loss / m  print(f"Epoch {epoch + 1}: Loss = {avg\_loss}")  loss\_history.append(avg\_loss)  return loss\_history;  def predict(self, x):  \_, \_, \_, output = self.forward\_propagation(x)  predictions = np.argmax(output, axis=1)  return predictions  nn = HardcodedNeuralNetwork(2, 3, 1)  num = 100  X = []  y = []  for i in np.arange(0,num):  x1 = np.random.uniform(low=0, high=1)  x2 = np.random.uniform(low=0, high=1)  X.append((x1, x2))  y.append(x1+x2)  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  X\_train = np.array(X\_train)  X\_test = np.array(X\_test)  y\_train = np.array(y\_train)  y\_test = np.array(y\_test)  X\_train = X\_train.T  X\_test = X\_test.T  X\_train = X\_train.reshape(2, -1)  X\_test = X\_test.reshape(2, -1)  y\_train = y\_train.reshape(1, -1)  y\_test = y\_test.reshape(1, -1)  history = nn.train(X\_train, y\_train, 100, 0.01)  loss = np.array(history)  epoch = np.arange(0,len(loss))  plt.plot(epoch, loss,label='loss')  plt.xlabel('epoch')  plt.ylabel('total loss')  plt.show() |

****

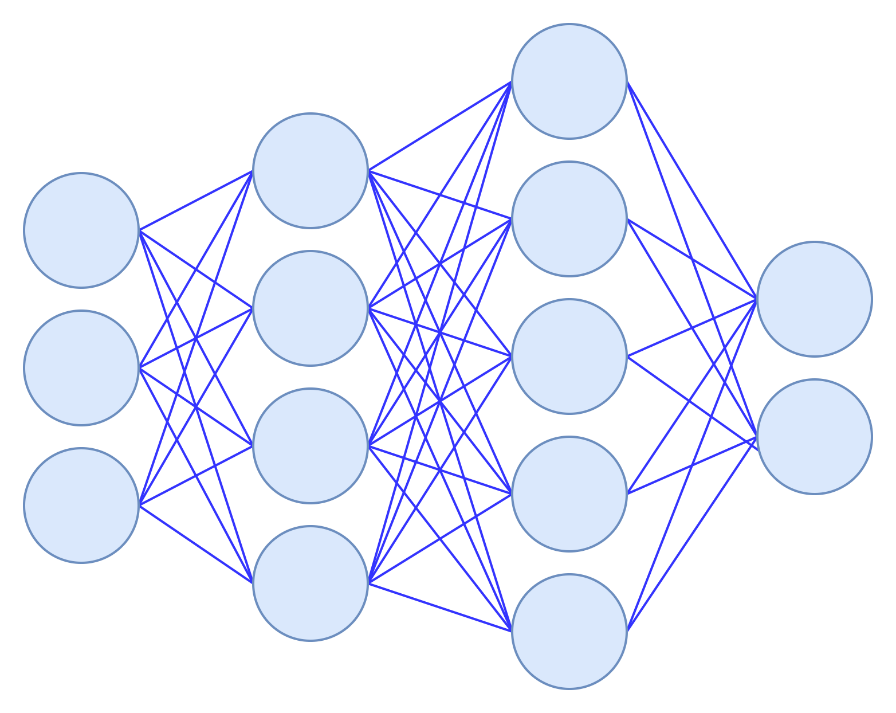
**Discussion of results:**

Results show a consistent decrease in the loss across the epochs (from 0.288 in epoch 1 to 0.168 in epoch 100), meaning that the training of the neural network works. However, the loss value of 0.168 in the final epoch is still relatively high. This could be due to a variety of reasons. One reason could be due to the choice of sigmoid as the activation function for the output layer. Since our inputs x1 and x2 are both between (0,1), and our output y = x1 + x2, y should be between (0,2). However, the sigmoid function is bounded between (0,1) so predictions that are supposed to be between (1,2) are always reduced 1, making the loss higher in some cases.

**Question 2: Backpropagation for a Deep Neural Network by Hand (4 marks)**

Objective of this question is to do one forward, backward step using backpropagation based on the matrix notation discussed in the lecture “Implementing Backpropogation”.

Use the following deep neural network with multiple hidden layers. Assume all weights are 0.1, all biases are 0.1also. Use leaky ReLu with alpha = 0.1 for all layers except the final layer which is Sigmoid. Use MSE as the loss function. The network architecture is as follows:



|  |  |
| --- | --- |
| **Input Data** | **Target Output** |
| [0.3, -0.1, 0.8] | [1, 0] |
| [-0.2, 0.3, 0.5] | [1, 0] |
| [0.5, -0.1, 0.9] | [0, 1] |
| [0.1, 0.3, -0.1] | [0, 1] |
| [0.2, 0.8, -0.2] | [1,0] |

Given the input data and their corresponding target outputs, perform the following:

a) Perform the forward pass to calculate the predicted output of the network. Show all intermediate steps including all matrix multiplications and SCHUR products.

b) Calculate the loss using the MSE loss function.

c) Perform the backward pass (backpropagation) to compute the gradients of the weights and biases for each layer. Show all calculations and intermediate steps.

d) Update the weights and biases using a learning rate of 0.01.

**This question needs to be done by hand. Feel free to use code to test your answer.**

A close-up of a white paper

Description automatically generatedA close-up of a paper

Description automatically generatedA white paper with writing on it

Description automatically generatedA close-up of a white paper

Description automatically generatedA close-up of a paper

Description automatically generatedA close-up of a white paper

Description automatically generated

**Question 3: Backpropagating through the Input (2 marks)**

The objective of this question is to investigate what happens if we continue to calculate dL/dz for the input data.

Show how this can be done for Question 2. Explain why this is a good idea or not and if so, then how can this be used?

Calculating the derivative of the loss function with respect to the input data shows how input features impact the model's performance. This can be helpful in understanding how sensitives the model is to differing inputs in turn this can be used for feature engineering to understand what features are useful and which are not. On the other hand, this increases the risk of overfitting because we're training the model on specific input sets which won't translate well for the testing set.