Table of Contents

[Artificial Neural Networks (ANN) Introduction 1](#_Toc156573254)

[Forward Pass 4](#_Toc156573255)

[Applying Weights and Bias of the Neurons in the Hidden Layer 4](#_Toc156573256)

[Applying the Activation Function for the Hidden Layer 4](#_Toc156573257)

[Applying Weights and Bias of the Neurons in the Output Layer 5](#_Toc156573258)

[Applying the Activation Function for the Output Layer 6](#_Toc156573259)

[Back Propagation 7](#_Toc156573260)

[Calculate the losses 7](#_Toc156573261)

[TensorFlow and Keras 9](#_Toc156573262)

[PyTorch 13](#_Toc156573263)

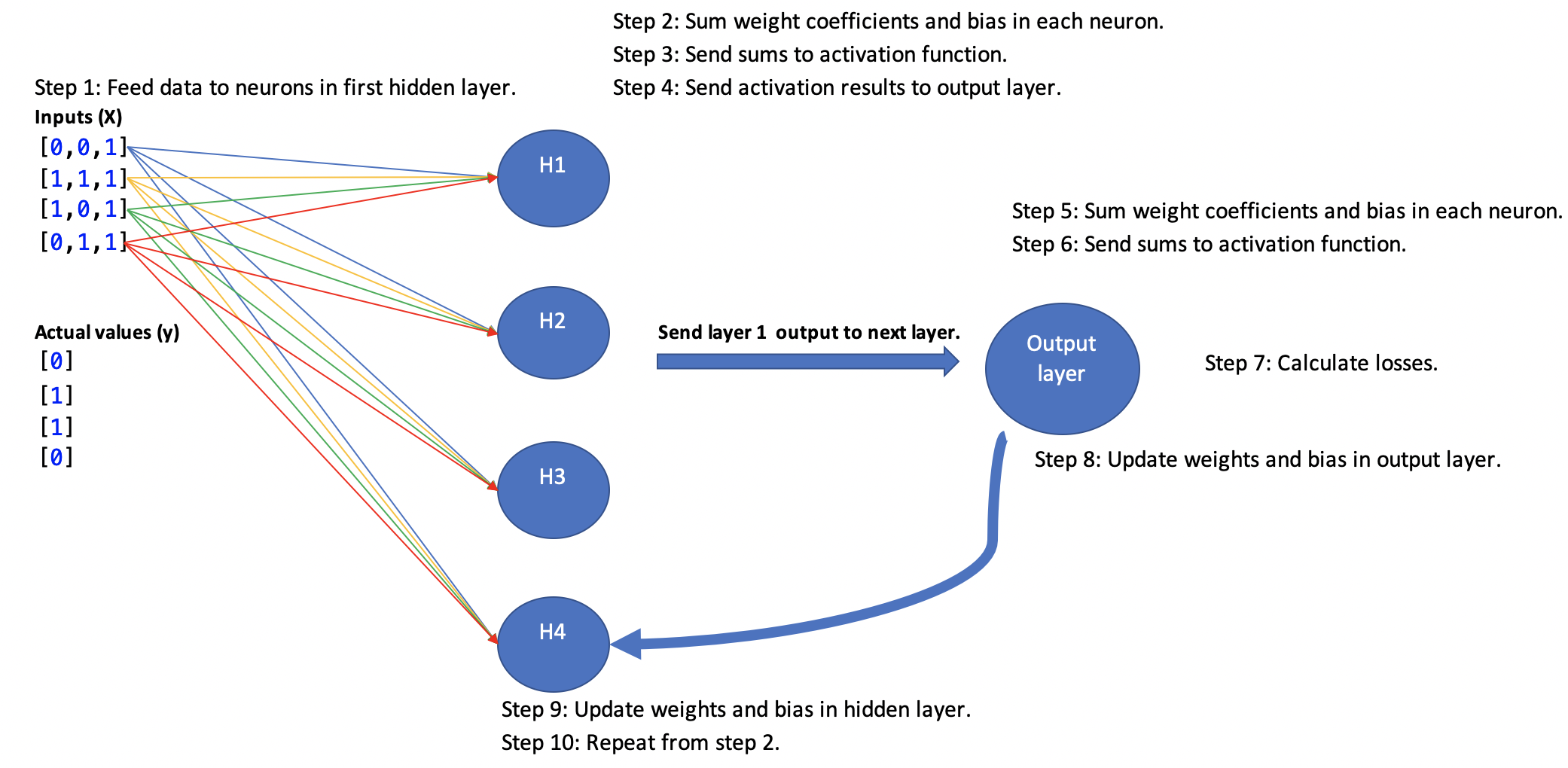
# Artificial Neural Networks (ANN) Introduction

This document introduces the topic of artificial neural networks. The steps for ANN learning are summarized in Figure 1.

Some important items to note about the network are:

* Each sample is fed to each neuron of the hidden and output layers.
* Each neuron stores its own weights and bias.
* The activation function for this network is a simple sigmoid function.
* A gradient descent algorithm is used to calculate losses.

Figure 1: Neural Network Summary of Learning



Example 1: ANN from Scratch

This example shows the code needed to implement the network that is summarized in Figure 1. The network learns with the X and y values in the table and is able to accurately predict y when finished training.

|  |  |  |
| --- | --- | --- |
| y | X |  |
| Actual data:  [[0]  [1]  [1]  [0]] | [[0, 0, 1],  [1, 1, 1],  [1, 0, 1],  [0, 1, 1]] | ANN predictions:  [[0.02123225]  [0.98112875]  [0.98349298]  [0.01860364]] |

Here is the code:

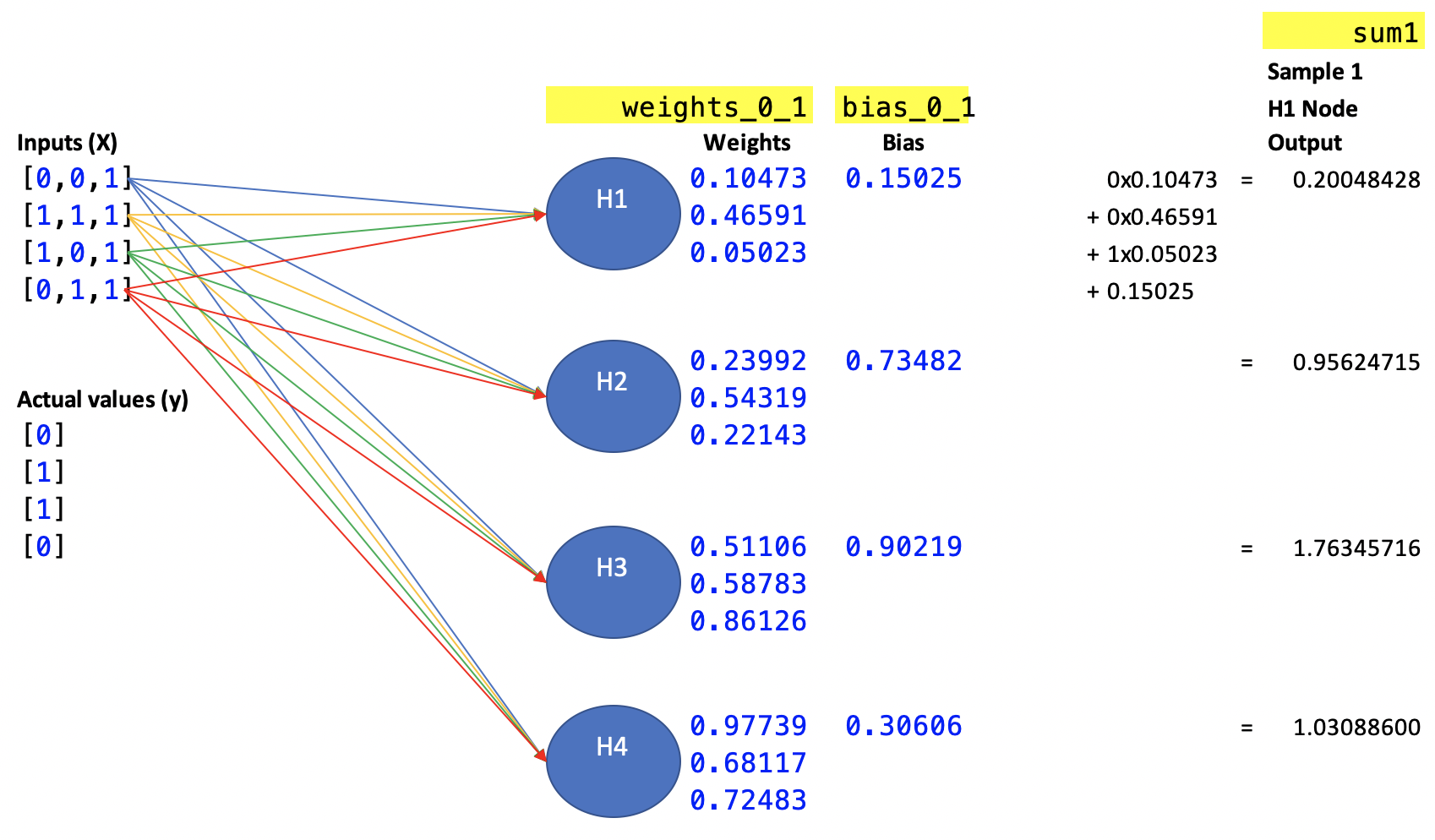
|  |
| --- |
| # importing libraries  import numpy as np  import pandas as pd  ### DATA #######################################  # input data  # (no scaling required in this case for simplicity)  X\_train\_scaled = np.array([[0, 0, 1],  [1, 1, 1],  [1, 0, 1],  [0, 1, 1]])  # output data  y\_train = np.array([ [0],  [1],  [1],  [0]])  ### DATA #######################################  # defining sigmoid activation function  def sigmoid(x):  return 1 / (1 + np.exp(-x))  # defining derivative of sigmoid activation function  def sigmoid\_derivative(x):  return x \* (1 - x)  # Pass data through layers.  # Each layer sums applied weights and bias, passes to  # an activation function.  def feedForward(X\_scaled):  sum1 = np.dot(X\_scaled, weights\_0\_1) + bias\_0\_1  sum1 = np.array(sum1)  layer\_1 = sigmoid(sum1) # Activation function.  sum2 = np.dot(layer\_1, weights\_1\_2) + bias\_1\_2  layer\_2 = sigmoid(sum2) # Activation function.  return layer\_1, layer\_2  # Update weights and bias from front to back.  def backPropogate(weights\_0\_1, bias\_0\_1, weights\_1\_2, bias\_1\_2):  yDf = pd.DataFrame(data=y\_train, columns=['admitted'])  # Calculate prediction error.  layer\_2\_error = layer\_2 - np.array(yDf['admitted']).reshape(-1, 1)  # Get rate of change of cost function.  layer\_2\_delta = layer\_2\_error \* sigmoid\_derivative(layer\_2)    # Determine layer 1 error as cost rate of change \* layer 2 weights.  layer\_1\_error = layer\_2\_delta.dot(weights\_1\_2.T)  # Get rate of change for layer 1.  layer\_1\_delta = layer\_1\_error \* sigmoid\_derivative(layer\_1)  # Update weights and bias.  weights\_1\_2 -= layer\_1.T.dot(layer\_2\_delta) \* learning\_rate  weights\_0\_1 -= layer\_0.T.dot(layer\_1\_delta) \* learning\_rate  bias\_1\_2 -= np.sum(layer\_2\_delta, axis=0, keepdims=True) \* learning\_rate  bias\_0\_1 -= np.sum(layer\_1\_delta, axis=0, keepdims=True) \* learning\_rate  return weights\_0\_1, bias\_0\_1, weights\_1\_2, bias\_1\_2  # defining learning rate  learning\_rate = 0.1  # These weights would normally be generated randomly  # or with kernel initializers.  weights\_0\_1 = np.array([[0.10473281, 0.23991864, 0.51106061, 0.97739018],  [0.46591006, 0.54318817, 0.58782883, 0.68117129],  [0.0502301, 0.22142866, 0.86126238, 0.72482657]])  bias\_0\_1 = np.array([[0.15025418, 0.73481849, 0.90219478, 0.30605943]])  weights\_1\_2 = np.array([[0.62996657], [0.25984049], [0.72180012], [0.81730325]])  bias\_1\_2 = np.array([[0.09842751]])  # training loop  EPOCHS = 10000  for i in range(EPOCHS):  layer\_0 = X\_train\_scaled  # Feed data forward.  layer\_1, layer\_2 = feedForward(X\_train\_scaled)  # Back propagate to update weights and bias.  weights\_0\_1, bias\_0\_1, weights\_1\_2, bias\_1\_2 = \  backPropogate(weights\_0\_1, bias\_0\_1, weights\_1\_2, bias\_1\_2)  # printing output  print('Actual data:')  print(y\_train)  print('ANN predictions:')  print(layer\_2) |

## Forward Pass

### Applying Weights and Bias of the Neurons in the Hidden Layer

Figure 2 shows how the weights and bias of a single neuron are applied to the first sample during a forward pass through the network.

Figure 2: Weights and Bias of each Neuron are Applied to First Sample Where X=[0,0,1]



Exercise 1 (5 marks)

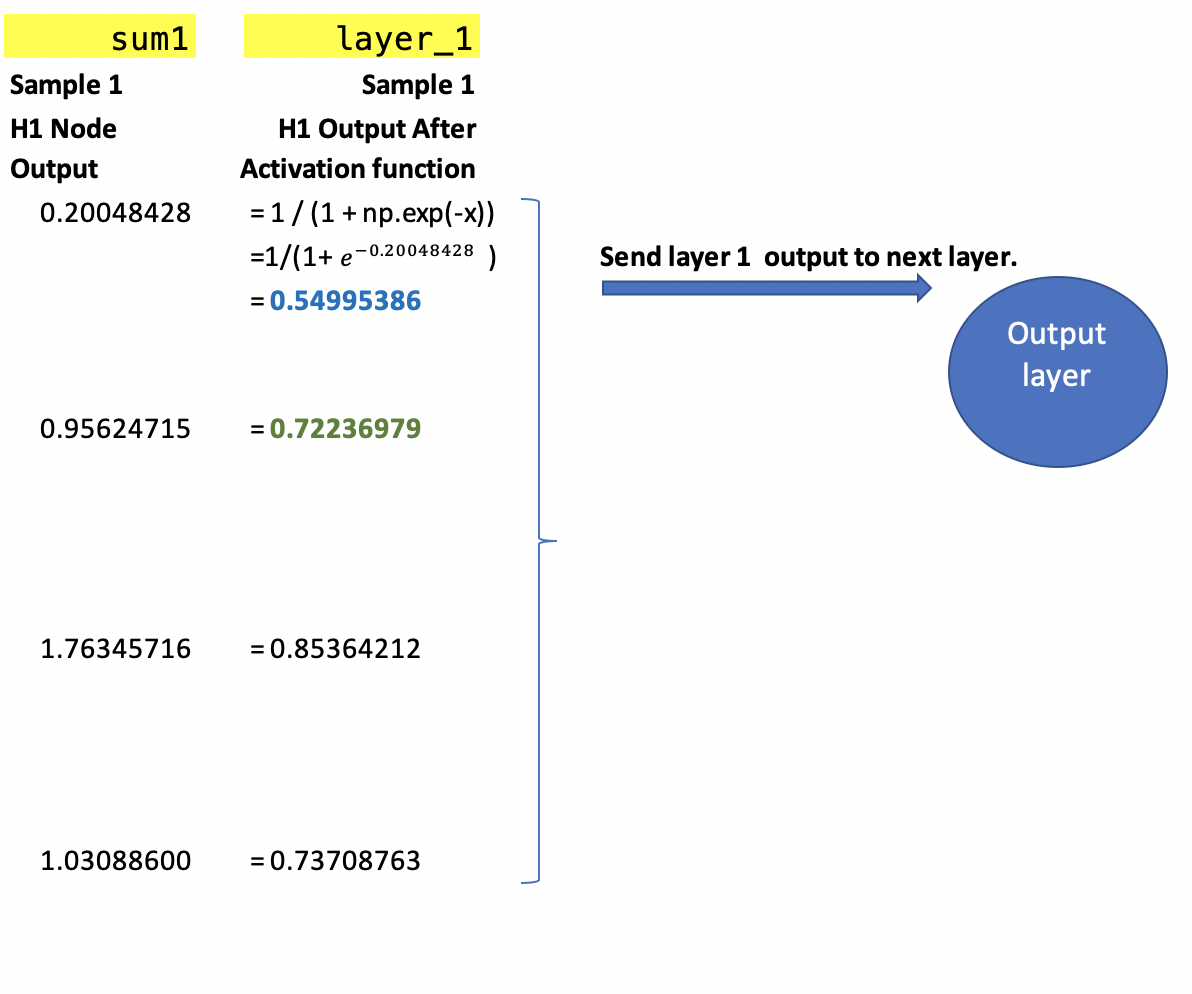
Show calculations that are applied by the second neuron, H2, to sum the respective weights\*features plus bias for the second sample where X = [1,1,1]. Show the final sum as well.   
Note that it is possible to step through the code to check your work.

|  |
| --- |
|  |

### Applying the Activation Function for the Hidden Layer

After weights and biases are applied to each of the samples, their sums are passed to the activation function. The activation function used in this example happens to be a sigmoid function. After the sums are passed through the sigmoid function the data is then passed to the next layer. (See Figure 3).

Figure 3: Weight and bias sums of each neuron are passed to the activation function. Activation results are sent to next layer.



Exercise 2 (3 marks)

Show the array of outputs from the activation function for the second sample where X=[1,1,1]. It is possible to check your work by stepping through the code. There should be four elements in this output for the second sample.

|  |
| --- |
|  |

Exercise 3 (2 marks)

Show the activation function equation with any necessary value(s) plugged in to calculate the first value in the array for Exercise 2.

|  |
| --- |
|  |

Exercise 4 (1 mark)

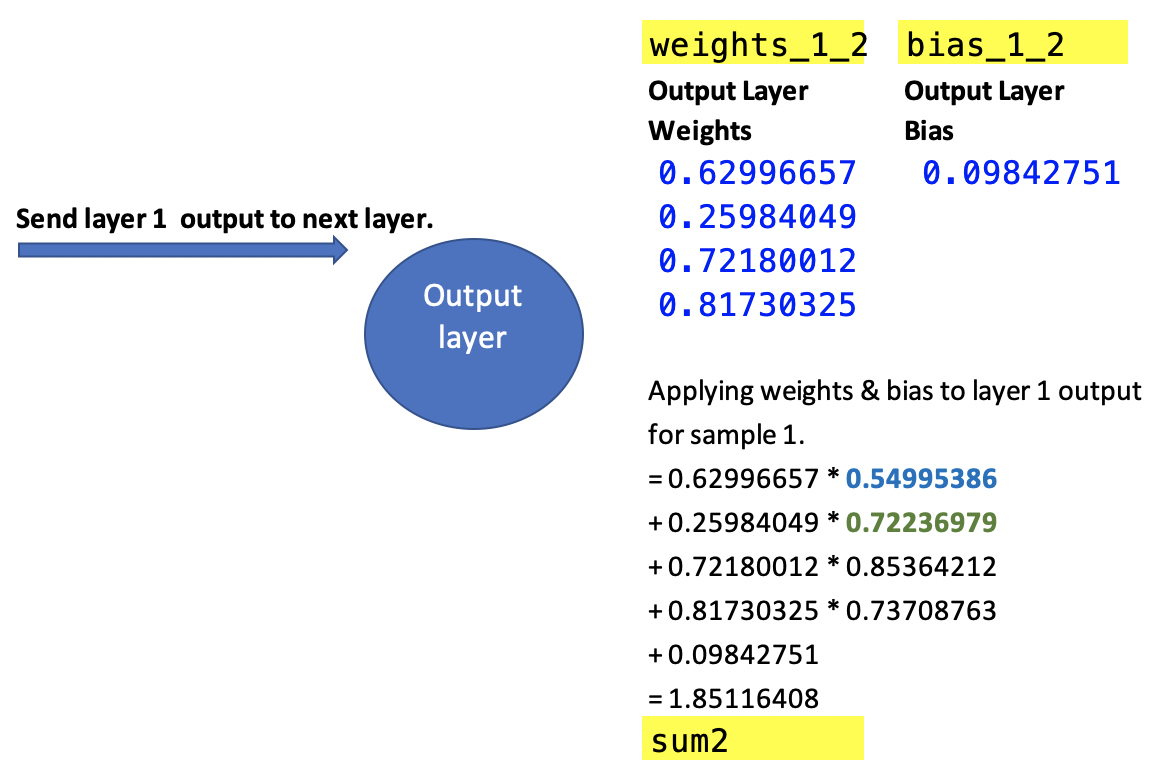
Show a picture of the graph for a sigmoid function here:

|  |
| --- |
|  |

### Applying Weights and Bias of the Neurons in the Output Layer

This step shows how the output layer weights and bias are applied to the data that is sent from the previous layer.

Figure 4: Weights and bias are applied to sample outputs from the previous layer.



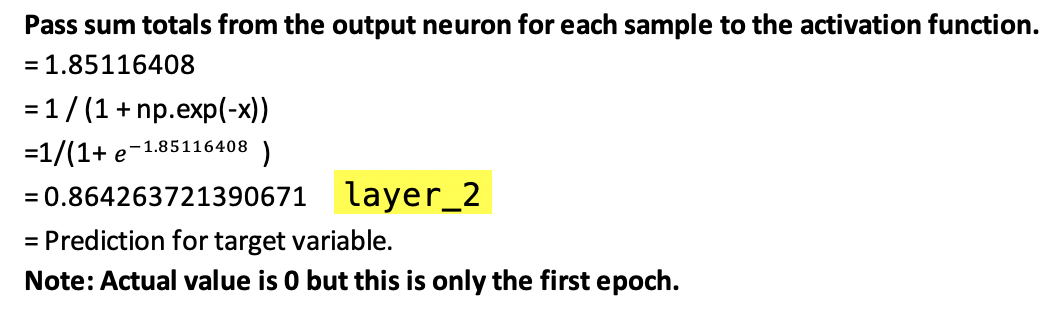
Exercise 5 (5 marks)

Using the sample 2 output from the activation function of the hidden layer where the input X = [1, 1, 1]. Apply the weights and bias of the output layer and show the calculations and resulting sum. Show the calculation. (You can check your work by stepping through the code and examining sum2).

|  |
| --- |
|  |

### Applying the Activation Function for the Output Layer

In the last step of the forward pass, the sums from the output neuron are fed to the activation function. The resulting value is the prediction.



Exercise 6 (2 marks)

After the first epoch, what is the prediction for the second sample where input X is [1,1,1]?

|  |
| --- |
|  |

What is the actual target value for the second sample?

|  |
| --- |
|  |

Exercise 7 (2 marks)

Add an instruction to print out the sample predictions after each weight and bias update at each epoch.

print(layer\_2)

Show the predictions after the first epoch here. Hint, the prediction for the first sample is 0.86426372.

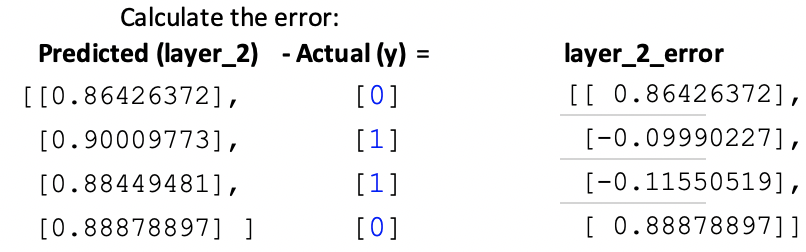
|  |
| --- |
|  |

## Back Propagation

After each forward pass, back propagation updates the weights and bias of each neuron using a gradient descent approach to move towards the optimum.

### Calculate the losses

First losses are calculated. The losses are measured in this case by the distance between actual and predicted values.



Exercise 8 (3 marks)

Show the layer\_2\_error calculations at the last epoch here:

|  |
| --- |
|  |

Is there a noticeable reduction in layer\_2\_error at the last epoch compared to the first?

|  |
| --- |
|  |

Exercise 9 (5 marks)

Complete the table to help explain how accuracy improves during iterative learning in Example 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Actual y | Predictions after 100 epochs | Predictions after 1000 epochs | Predictions after 10000 epochs |
| [[0]  [1]  [1]  [0]] |  |  |  |

What conclusion can be made about accuracy and learning for this network?

|  |
| --- |
|  |

Example 2: Neural Network Applied

This example shows how to use the network from Example 1 to build a model which predicts whether a student will be accepted to a college or University where they apply. To build this example, replace the section that is surrounded by the open and close ## Data ## comment with the following.

|  |
| --- |
| ### DATA #######################################  # Setup data.  candidates = {'gmat': [780,750,690,710,680,730,690,720,  740,690,610,690,710,680,770,610,580,650,540,590,620,  600,550,550,570,670,660,580,650,660,640,620,660,660,  680,650,670,580,590,690],  'gpa': [4,3.9,3.3,3.7,3.9,3.7,2.3,3.3,  3.3,1.7,2.7,3.7,3.7,3.3,3.3,3,2.7,3.7,2.7,2.3,  3.3,2,2.3,2.7,3,3.3,3.7,2.3,3.7,3.3,3,2.7,4,  3.3,3.3,2.3,2.7,3.3,1.7,3.7],  'work\_experience': [3,4,3,5,4,6,1,4,5,  1,3,5,6,4,3,1,4,6,2,3,2,1,4,1,2,6,4,2,6,5,1,2,4,6,  5,1,2,1,4,5],  'admitted': [1,1,1,1,1,1,0,1,1,0,0,1,  1,1,1,0,0,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,1,1,1,0,0,  0,0,1]}  df = pd.DataFrame(candidates,columns= ['gmat', 'gpa',  'work\_experience','admitted'])  y = np.array(df['admitted'])  X = df.copy()  del X['admitted']  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import StandardScaler  # Split data.  X\_train,X\_test,y\_train,y\_test = train\_test\_split(X, y, test\_size=0.25,  random\_state=0)  # define standard scaler  scaler = StandardScaler()  # transform data  X\_train\_scaled = scaler.fit\_transform(X\_train)  X\_test\_scaled = scaler.transform(X\_test)  ### DATA ####################################### |

Exercise 10 (5 marks)

ANN’s are sensitive to scaling. Most of the time inputs should be scaled. Modify Example 2 by removing the scaling. What predictions are made with the training data when all epochs have been completed?

|  |
| --- |
|  |

Step through the code in Example 2and identify the point where the algorithm fails with unscaled data. Explain what went wrong. Please use your best judgement here – there is no single perfect way to answer this question but you should be able to observe some point early on when the algorithm is unable to learn.

|  |
| --- |
|  |

Example 3: Making Predictions

This example shows that the network is trained up perfectly to make accurate predictions with the test data from Example 2.

|  |
| --- |
| [0 0 1 1 0 0 1 1 0 1]  precision recall f1-score support  0 1.00 1.00 1.00 5  1 1.00 1.00 1.00 5  accuracy 1.00 10  macro avg 1.00 1.00 1.00 10  weighted avg 1.00 1.00 1.00 10 |

To build this example add this code to the end of Example 2. The neural network has trained up properly.

|  |
| --- |
| from sklearn.metrics import classification\_report  # Make predictions.  layer\_1, layer\_2 = feedForward(X\_test\_scaled)  print(layer\_2)  print(y\_test)  predictions = []  for i in range(0, len(layer\_2)):  if(layer\_2[i]>0.5):  predictions.append(1)  else:  predictions.append(0)  print(classification\_report(y\_test, predictions)) |

## TensorFlow and Keras

I have been using Tensorflow and Keras. The book by Dr. Brownlee also uses Keras. I like Keras because it provides structures that are easy to learn.

Example 4: Building a Sequential ANN with Keras

This example shows how to build an ANN to predict college admission acceptance like in the earlier examples in this document. To run this code likely you will need to install tensorflow and keras with pip.

When running the code, the performance appears to be acceptable.

|  |
| --- |
| Actual:  [0 0 1 1 0 0 1 1 0 1]  Predicted:  [[0.0219794 ]  [0.02577448]  [0.98109984]  [0.98109984]  [0.02254776]  [0.02202809]  [0.98113185]  [0.981141 ]  [0.0219158 ]  [0.980833 ]]  precision recall f1-score support  0 1.00 1.00 1.00 5  1 1.00 1.00 1.00 5  accuracy 1.00 10  macro avg 1.00 1.00 1.00 10  weighted avg 1.00 1.00 1.00 10 |

The network built is a 10 neuron x 3 neuron x 1 neuron or 10x3x1 ANN.

Here is the code:

|  |
| --- |
| # importing libraries  import numpy as np  import pandas as pd  ### DATA #######################################  # Setup data.  candidates = {'gmat': [780,750,690,710,680,730,690,720,  740,690,610,690,710,680,770,610,580,650,540,590,620,  600,550,550,570,670,660,580,650,660,640,620,660,660,  680,650,670,580,590,690],  'gpa': [4,3.9,3.3,3.7,3.9,3.7,2.3,3.3,  3.3,1.7,2.7,3.7,3.7,3.3,3.3,3,2.7,3.7,2.7,2.3,  3.3,2,2.3,2.7,3,3.3,3.7,2.3,3.7,3.3,3,2.7,4,  3.3,3.3,2.3,2.7,3.3,1.7,3.7],  'work\_experience': [3,4,3,5,4,6,1,4,5,  1,3,5,6,4,3,1,4,6,2,3,2,1,4,1,2,6,4,2,6,5,1,2,4,6,  5,1,2,1,4,5],  'admitted': [1,1,1,1,1,1,0,1,1,0,0,1,  1,1,1,0,0,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,1,1,1,0,0,  0,0,1]}  df = pd.DataFrame(candidates,columns= ['gmat', 'gpa',  'work\_experience','admitted'])  y = np.array(df['admitted'])  X = df.copy()  del X['admitted']  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import StandardScaler  # Split data.  X\_train,X\_test,y\_train,y\_test = train\_test\_split(X, y, test\_size=0.25,  random\_state=0)  # define standard scaler  scaler = StandardScaler()  # transform data  X\_train\_scaled = scaler.fit\_transform(X\_train)  X\_test\_scaled = scaler.transform(X\_test)  ##########################################################  from tensorflow.keras import Sequential  from tensorflow.keras.layers import Dense  # Build a network model of sequential layers.  model = Sequential()  NUM\_COLS = 3  # Add 1st hidden layer. Note 1st hidden layer also receives data.  # The input array must contain two feature columns and any number of rows.  model.add(Dense(10, activation='sigmoid',  input\_shape=(NUM\_COLS,)))  # Add 2nd hidden layer.  model.add(Dense(3, activation='sigmoid'))  # Add output layer.  model.add(Dense(1, activation='sigmoid'))  # Compile the model.  # Binary cross entropy is used to measure error cost for binary predictions.  model.compile(loss='binary\_crossentropy', metrics=['accuracy'])  # Fit the model.  # An epoch is one iteration for all samples through the network.  # verbose can be set to 1 to show detailed output during training.  model.fit(X\_train\_scaled, y\_train, epochs=2000, verbose=1)  # Evaluate the model.  loss, acc = model.evaluate(X\_test\_scaled, y\_test, verbose=0)  print('Test Accuracy: %.3f' % acc)  # Make predictions.  yhats = model.predict(X\_test\_scaled)  print("Actual:")  print(y\_test)  print("Predicted: ")  print(yhats)  predictions = []  from sklearn.metrics import classification\_report  def showClassificationReport(y\_test, yhats):  # Convert continous predictions to  # 0 or 1.  for i in range(0, len(yhats)):  if(yhats[i]>0.5):  predictions.append(1)  else:  predictions.append(0)  print(classification\_report(y\_test, predictions))  showClassificationReport(y\_test, yhats) |

Exercise 11 (1 mark)

How many epochs are used in the network above?

|  |
| --- |
|  |

Exercise 12 (1 mark)

What important loss trend can you identify when observing the output while the network is learning?

|  |
| --- |
|  |

Exercise 13 (3 marks)

Modify Example 4 so it can predict if a flower is red or white. Here is the data:

|  |
| --- |
| # Load the flower feature data into a DataFrame.  df = pd.DataFrame(columns=['Length', 'Width', 'IsRed'])  data = [  {'Length':3, 'Width':1.5, 'IsRed': 1},  {'Length':2, 'Width':1, 'IsRed': 0},  {'Length':4, 'Width':1.5, 'IsRed': 1},  {'Length':3, 'Width':1, 'IsRed': 0},  {'Length':3.5, 'Width':.5, 'IsRed': 1},  {'Length':2, 'Width':.5, 'IsRed': 0},  {'Length':5.5, 'Width':1, 'IsRed': 1},  {'Length':1, 'Width':1, 'IsRed': 0},  {'Length':4.5, 'Width':1, 'IsRed':1}]  df = pd.DataFrame.from\_records(data) |

Show a screenshot of the output from your edited network here:

|  |
| --- |
|  |

## PyTorch

Example 5: College Admissions Using PyTorch

This example shows how to implement a model for classifying if applicants will be accepted to College by using PyTorch.

This model has two fully connected layers, with 3 input nodes and 8 hidden nodes, and an output node that gives the probability of the input belonging to class 1. The model is trained using stochastic gradient descent and the binary cross-entropy loss function. The accuracy of the model is computed on the sample data and is printed out at the end. Note that the sample data is random, so the accuracy may be different each time the model is run.

|  |
| --- |
| import torch  import torch.optim as optim  from sklearn.model\_selection import train\_test\_split  ### DATA #######################################  # Setup data.  import pandas as pd  import numpy as np  candidates = {'gmat': [780,750,690,710,680,730,690,720,  740,690,610,690,710,680,770,610,580,650,540,590,620,  600,550,550,570,670,660,580,650,660,640,620,660,660,  680,650,670,580,590,690],  'gpa': [4,3.9,3.3,3.7,3.9,3.7,2.3,3.3,  3.3,1.7,2.7,3.7,3.7,3.3,3.3,3,2.7,3.7,2.7,2.3,  3.3,2,2.3,2.7,3,3.3,3.7,2.3,3.7,3.3,3,2.7,4,  3.3,3.3,2.3,2.7,3.3,1.7,3.7],  'work\_experience': [3,4,3,5,4,6,1,4,5,  1,3,5,6,4,3,1,4,6,2,3,2,1,4,1,2,6,4,2,6,5,1,2,4,6,  5,1,2,1,4,5],  'admitted': [1,1,1,1,1,1,0,1,1,0,0,1,  1,1,1,0,0,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,1,1,1,0,0,  0,0,1]}  df = pd.DataFrame(candidates,columns= ['gmat', 'gpa',  'work\_experience','admitted'])  y = np.array(df['admitted'])  X = df.copy()  del X['admitted']  X = X  ##########################################################  # Split the data into training and test sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)  # define standard scaler  from sklearn.preprocessing import StandardScaler  scaler = StandardScaler()  # transform data  X\_train\_scaled = scaler.fit\_transform(X\_train)  X\_test\_scaled = scaler.transform(X\_test)  # Convert the data to PyTorch tensors  X\_train = torch.tensor(X\_train\_scaled, dtype=torch.float32)  X\_test = torch.tensor(X\_test\_scaled, dtype=torch.float32)  # Reshapes array.  # unsqueeze() creates array of single dimensional arrays.  y\_train = torch.tensor(y\_train, dtype=torch.float32).unsqueeze(1)  y\_test = torch.tensor(y\_test, dtype=torch.float32).unsqueeze(1)  import torch  import torch.nn as nn  # Define the neural network architecture  class BinaryClassificationNet(nn.Module):  def \_\_init\_\_(self):  super(BinaryClassificationNet, self).\_\_init\_\_()  self.fc1 = nn.Linear(3, 8)  self.fc2 = nn.Linear(8, 1)  self.sigmoid = nn.Sigmoid()  def forward(self, x):  x = self.fc1(x) # Hidden layer.  x = self.sigmoid(x) # Activation function.  x = self.fc2(x) # Output layer.  x = self.sigmoid(x) # Activation function.  return x  # Instantiate the model  model = BinaryClassificationNet()  # Define the loss function and optimizer  criterion = nn.BCELoss() # Binary cross entropy.  # Use stochastic gradient descent to update weights & bias.  optimizer = optim.SGD(model.parameters(), lr=0.01)  # Train the model  for epoch in range(2000):  print("Epoch: " + str(epoch))  # Forward pass  output = model(X\_train)  loss = criterion(output, y\_train)  # Backward pass and optimization  optimizer.zero\_grad()  loss.backward()  optimizer.step()  # Evaluate the model  with torch.no\_grad():  outputs = model(X\_test)  predictions = outputs.round()  accuracy = (predictions == y\_test).float().mean()  print(f'Accuracy: {accuracy}') |

Exercise 14 (6 marks)

Using the data from Exercise 13, modify Example 5 to use PyTorch with this data set. Also, add the classification report to show the precision, recall and accuracy scores with the test data. Show your complete program after making the changes here.

|  |
| --- |
|  |