Table of Contents

[Artificial Neural Networks 1](#_Toc92362505)

[Deep Learning 1](#_Toc92362506)

[Multi-Layer Perceptron (MLP) 1](#_Toc92362507)

[Manually Coding a Single Node Feedforward Neural Network 2](#_Toc92362508)

[TensorFlow and Keras 6](#_Toc92362509)

## Artificial Neural Networks

An artificial neural network (ANN) is a network of nodes. Each node processes a portion of the data but the efforts of all nodes are weighted and combined to perform classifications and regressions for continuous variables.

## Deep Learning

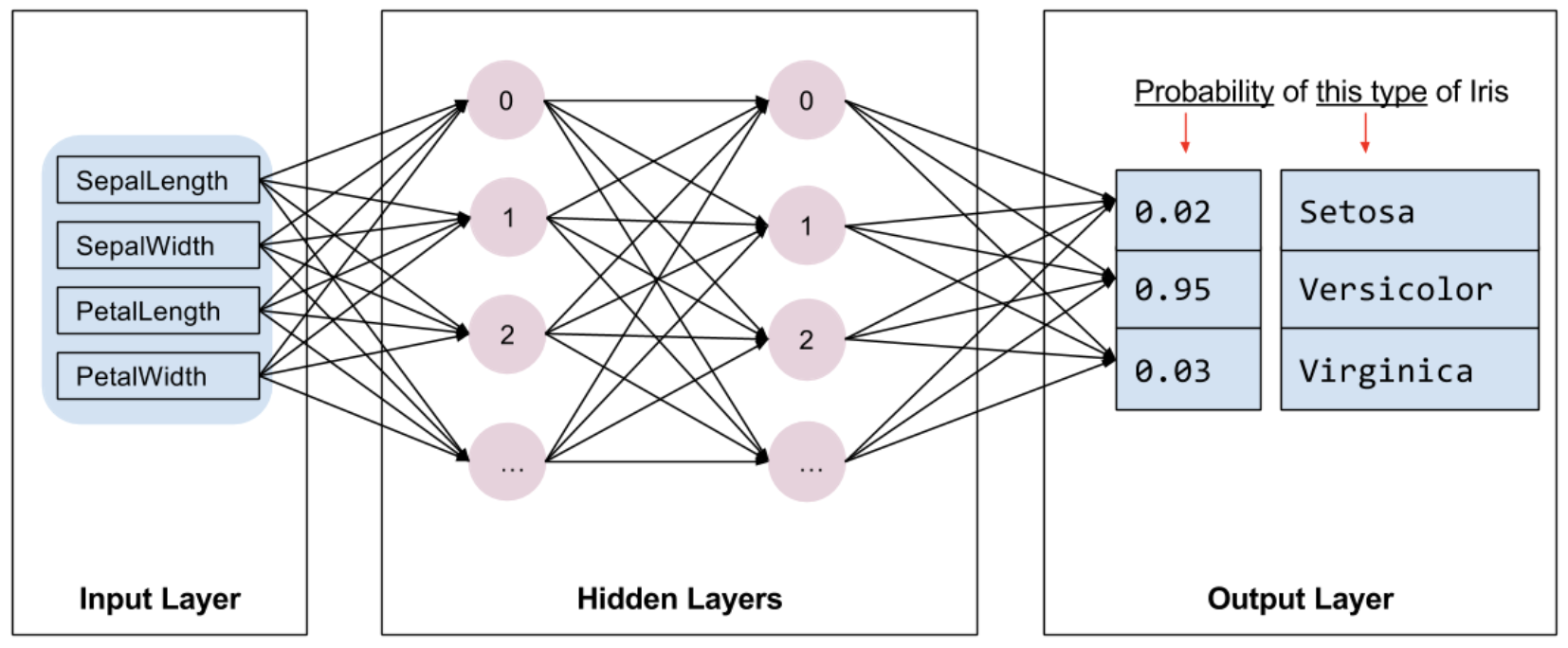
When we work with neural networks we are extending our machine learning algorithms to layered networks. You will often hear people refer to these algorithms as deep learning algorithms due to the inclusion of multiple hidden layers.

## Multi-Layer Perceptron (MLP)

This is a forward feed multi-layer perceptron network. All nodes, called **neurons**, are interconnected in this case. The data is fed through the network and classification predictions are the output.

Simple networks with linear relationships may require fewer layers. However, more complex relationships can be represented with deeper (more hidden layer) networks.

Figure 1: MLP Neural Network



## Manually Coding a Single Node Feedforward Neural Network

Example 1: Building a Single Neural Network Node

This example shows a feedforward neural network with back propagation. The loss (cost) is calculated using gradient descent. The gradient descent algorithm was discussed last day.

The example is based on code that is written during the following video series which is worth watching if you have time:

<https://www.youtube.com/watch?v=gwitf7ABtK8&list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So&index=4>

<https://www.youtube.com/watch?v=c6NBkkKNZXw&list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So&index=5>

<https://www.youtube.com/watch?v=Gvq9sUHPgrc&list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So&index=6>

<https://www.youtube.com/watch?v=EnGmg-kvpYs&list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So&index=7>

<https://www.youtube.com/watch?v=GwVTM28HKYk&list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So&index=8>

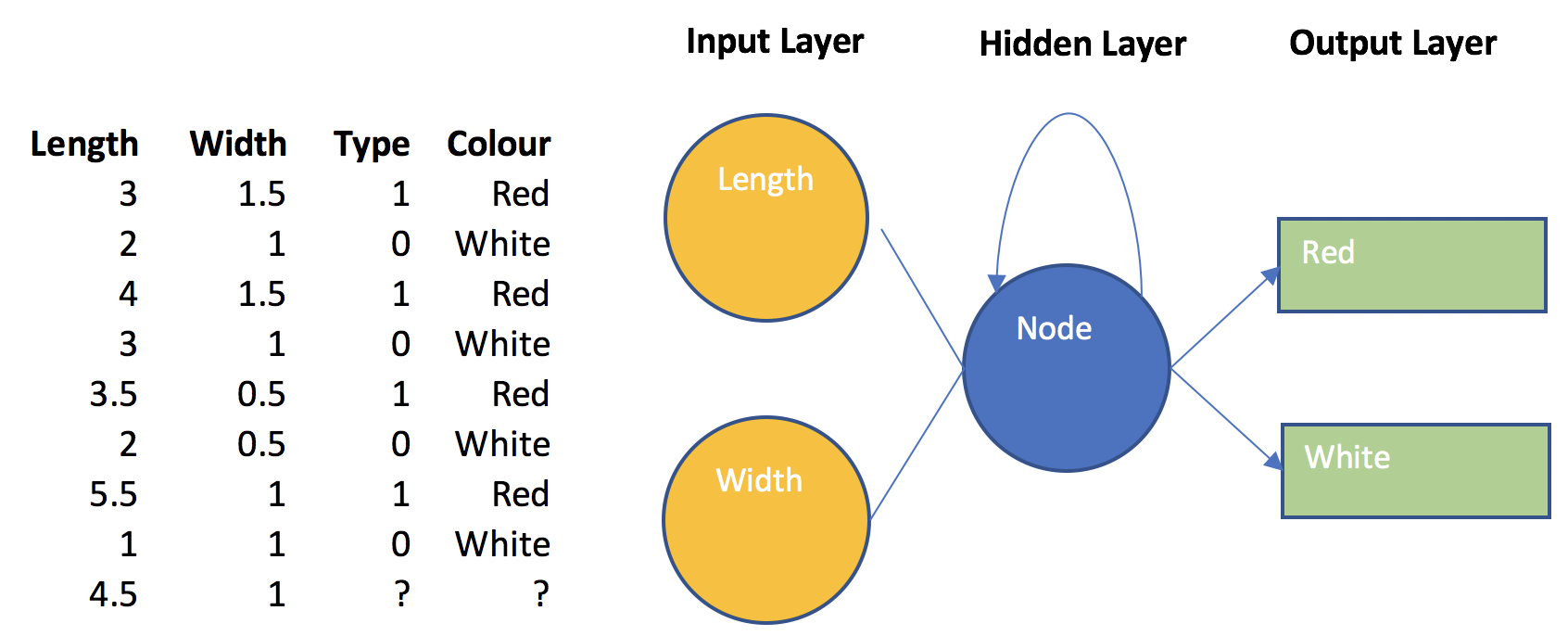
[**https://www.youtube.com/watch?v=fh1QZ97izSM&list=PLxt59R\_fWVzT9bDxA76AHm3ig0Gg9S3So&index=9**](https://www.youtube.com/watch?v=fh1QZ97izSM&list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So&index=9)

<https://www.youtube.com/watch?v=bP2SCk-QGkw&list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So&index=10>

<https://www.youtube.com/watch?v=gQLKufQ35VE&list=PLxt59R_fWVzT9bDxA76AHm3ig0Gg9S3So&index=11>

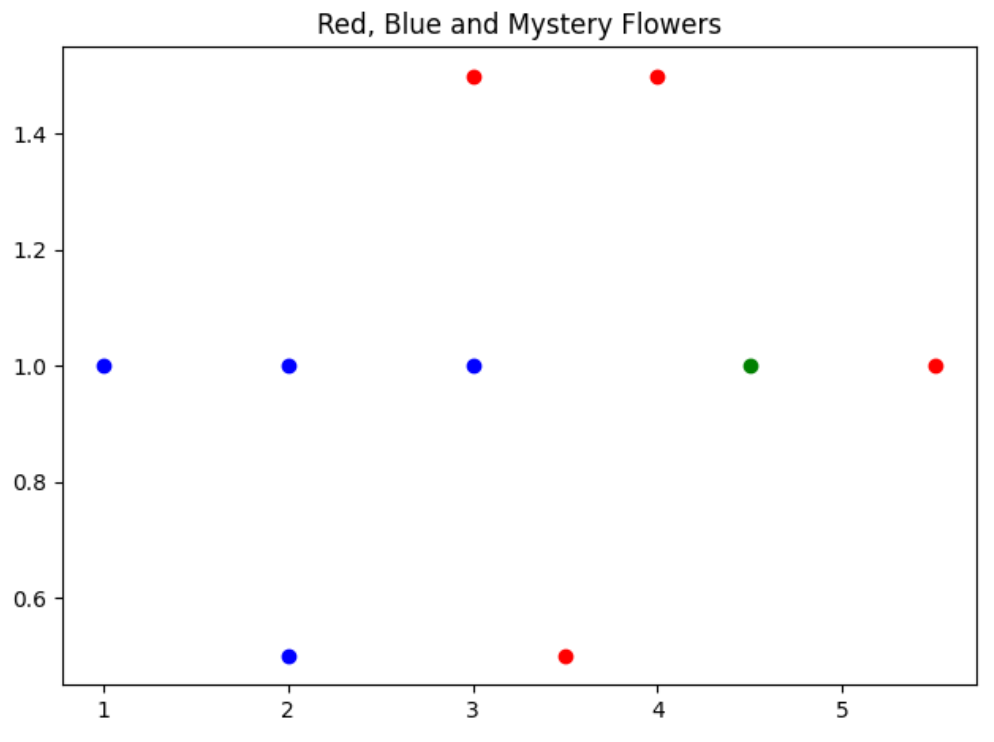
The network built in this example receives flower length and width to predict flower colour. Refer to Figure 2 to view the actual flower attribute values and the network diagram.

Figure 2: Data Inputs and Neural Network Diagram



If we plot the data we can see there is reasonable separation between the red and blue flowers. The mystery flower (see the green dot) appears to have attributes that fall within the red flower category (refer to Figure 3).

Figure 3; Scatter Plot of Flower Samples for Blue, Red and Mystery Flowers



Some short-cuts were taken so proper validation was not performed in this example. However, aside from the highlighted error, the output from running the program generally suggests that the final model is successful.

|  |
| --- |
| \*\*\*Predictions Using Training Data  [3, 1.5, 1] predicted color: 0.8983977639667545 red  [2, 1, 0] predicted color: 0.006042552646746413 blue  [4, 1.5, 1] predicted color: 0.9997461355333558 red  [3, 1, 0] predicted color: 0.7302794628250244 red  [3.5, 0.5, 1] predicted color: 0.9459351023673833 red  [2, 0.5, 0] predicted color: 0.0018580382044924844 blue  [5.5, 1, 1] predicted color: 0.99999991176948 red  [1, 1, 0] predicted color: 1.3649737587078628e-05 blue  \*\*\*Predictions Using Mystery Flower  [4.5, 1] predicted color: 0.9999607061817346 red |

This may be one of the last times that you will see a neural network that is coded from scratch during this course. Understanding the neural network now will give you more confidence when working with more advanced structures and code libraries later. Here is the code which uses the gradient descent loss function algorithm which was covered last day:

|  |
| --- |
| import matplotlib.pyplot as plt  import numpy as np  def visualizeData(data, mystery\_flower):  # EDA: Draw a scatter plot to visualize the data.  for i in range(len(data)):  point = data[i]  color = 'r'  if point[2] == 0:  color = 'b'  plt.scatter(point[0], point[1], c=color)  plt.scatter(mystery\_flower[0], mystery\_flower[1], color='green')  plt.title("Red, Blue and Mystery Flowers")  plt.show()  # Compress linear equation result to a floating-point value between 0 and 1.  def sigmoid(x):  return 1 / (1 + np.exp(-x))  def showLosses(losses):  plt.plot(losses)  plt.title("Loss")  plt.show()  def getOptimalWeightsAndBias(data):  # Initialize weights and bias.  w1 = np.random.randn()  w2 = np.random.randn()  b = np.random.randn()  costs = []  learning\_rate = 0.2  for i in range(10000):  # Get random element from data set.  ri = np.random.randint(len(data))  point = data[ri]  # Regression line for flowers.  z = w1\*point[0] + w2\*point[1] + b  # Force prediction to a floating point value between a value of 0 and 1.  pred = sigmoid(z)  target = point[2] # This is a 1 or 0.  # This is the residual square which is the loss function.  cost = (pred - target) \*\* 2  # This is the derivative of the cost function with respect to the prediction.  dcost\_pred = 2 \* (pred - target)  # Change in cost with respect to the change in the target.  dcost\_dz = dcost\_pred \* pred  # Change in cost with respect to change in weight 1.  dcost\_dw1 = dcost\_dz \* point[0]  # Change in cost with respect to change in weight 2.  dcost\_dw2 = dcost\_dz \* point[1]  # Update weights and bias.  w1 = w1 - learning\_rate \* dcost\_dw1  w2 = w2 - learning\_rate \* dcost\_dw2  b = b - learning\_rate \* dcost\_dz  # Calculate cost every 10 iterations.  if (i % 10 == 0):  cost = 0  point = data[ri]  # Calculate cost for current prediction  z = point[0] \* w1 + point[1] \* w2 + b  pred = sigmoid(z)  target = point[2]  cost += np.square(pred - target)  costs.append(cost)  showLosses(costs)  return w1, w2, b  def showPredictions(data, title, w1, w2, b):  print("\n\*\*\*Predictions Using " + title)  # Show predictions for each data point.  for i in range(len(data)):  point = data[i]  z = point[0] \* w1 + point[1] \* w2 + b  pred = sigmoid(z)  color = 'blue'  if(pred > 0.5):  color = 'red'  color = str(pred) + " " + color  print(str(point) + " predicted color: " + color)  # Input data.  data = [  [3, 1.5, 1],  [2, 1, 0],  [4, 1.5, 1],  [3, 1, 0],  [3.5, .5, 1],  [2, .5, 0],  [5.5, 1, 1],  [1, 1, 0]]  # Unknown value.  mystery\_flower = [4.5, 1]  w1, w2, b = getOptimalWeightsAndBias(data)  showPredictions(data, "Training Data", w1, w2, b)  showPredictions([mystery\_flower], "Mystery Flower", w1, w2, b) |

Exercise 1 (3 marks)

The following list describes the steps that are taken by the code in Example 1. However, the steps listed below are not in the proper order:

* The network performs a linear prediction by applying weights and bias to a sample.
* The weights and biases of the linear equation are then updated with the rates of change during the current epoch.
* These steps are then repeated.
* Rates of change for the cost function are then calculated.
* The squared residual, , is then calculated using the output of the sigmoid function to determine the cost.
* The linear prediction is fed to a sigmoid function to transform it to a value between 0 and 1.

Re-arrange the list so the instructions are in order. Look at the code to verify the proper sequence. Show your re-ordered steps here:

|  |
| --- |
| 1. The network performs a linear prediction by applying weights and bias to a sample. 2. The linear prediction is fed to a sigmoid function to transform it to a value between 0 and 1. 3. The squared residual, , is then calculated using the output of the sigmoid function to determine the cost. 4. Rates of change for the cost function are then calculated. 5. The weights and biases of the linear equation are then updated with the rates of change during the current epoch. 6. These steps are then repeated. |

Exercise 2 (1 mark)

Fill in the blank. In Example 1, weights and bias for the linear equation are updated each step by adjusting them with rates of change multiplied by a \_\_learning rate\_\_\_\_\_\_\_\_.

Exercise 3 (1 mark)

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

|  |
| --- |
|  |

Exercise 4 (1 mark)

What is input into the sigmoid function in Example 1? What is the output from the sigmoid function?

|  |
| --- |
| The regression line for flowers (z) is fed into the sigmoid function. A float between 0 and 1 is output from the sigmoid function. |

## TensorFlow and Keras

TensorFlow is the premier open-source deep learning framework. It is maintained by Google. Keras is an API that helps to simplify the use of TensorFlow.

Example 2: Building a Neural Network with tf.Keras

This example shows how to implement TensorFlow to develop a neural network model to perform classification predictions using the data from Example 1. For this current case, the data is stored in a data frame:

|  |
| --- |
| Length Width IsRed  0 3.0 1.5 1.0  1 2.0 1.0 0.0  2 4.0 1.5 1.0  3 3.0 1.0 0.0  4 3.5 0.5 1.0  5 2.0 0.5 0.0  6 5.5 1.0 1.0  7 1.0 1.0 0.0 |

We first need to isolate the predictor columns. This line of code extracts the first two columns for length and width into a separate data frame.

|  |
| --- |
| dfX = df.iloc[:, 0:2] |

We then need to express each row from our predictor data in a vertical array of features. This code transforms the X values into the required format.

|  |
| --- |
| ROW\_DIM = 0  COL\_DIM = 1  x\_array = dfX.values  x\_arrayReshaped = x\_array.reshape(x\_array.shape[ROW\_DIM],  x\_array.shape[COL\_DIM]) |

These are the X values after the transformation:

|  |
| --- |
| array([[3. , 1.5],  [2. , 1. ],  [4. , 1.5],  [3. , 1. ],  [3.5, 0.5],  [2. , 0.5],  [5.5, 1. ],  [1. , 1. ]]) |

The target column also needs to be in a vertical array as well so we need to reshape it in the same manner as the predictor columns:

|  |
| --- |
| dfY = df.iloc[:, 2:3]  y\_array = dfY.values  y\_arrayReshaped = y\_array.reshape(y\_array.shape[ROW\_DIM],  y\_array.shape[COL\_DIM]) |

After the transformation, the target variable column from the DataFrame becomes:

|  |
| --- |
| array([[1.],  [0.],  [1.],  [0.],  [1.],  [0.],  [1.],  [0.]]) |

This example shows how to predict flower colour by using length and width as inputs:

|  |
| --- |
| from tensorflow.keras import Sequential  from tensorflow.keras.layers import Dense  import numpy as np  import pandas as pd  # 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}]  for i in range(0, len(data)):  df = df.append(data[i], ignore\_index=True)  print(df)  ROW\_DIM = 0  COL\_DIM = 1  # Convert DataFrame columns to vertical columns of features (as mentioned earlier).  dfX = df.iloc[:, 0:2]  ROW\_DIM = 0  COL\_DIM = 1  x\_array = dfX.values  x\_arrayReshaped = x\_array.reshape(x\_array.shape[ROW\_DIM],  x\_array.shape[COL\_DIM])  # Convert DataFrame columns to vertical columns of target variables values.  dfY = df.iloc[:, 2:3]  y\_array = dfY.values  y\_arrayReshaped = y\_array.reshape(y\_array.shape[ROW\_DIM],  y\_array.shape[COL\_DIM])  # Build a network model of sequential layers.  model = Sequential()  # Add 1st hidden layer. Note 1st hidden layer also receives data from input layer.  # The input array must contain two feature columns and any number of rows.  model.add(Dense(10, activation='sigmoid',  input\_shape=(x\_arrayReshaped.shape[COL\_DIM],)))  # 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\_arrayReshaped, y\_arrayReshaped, epochs=1000, verbose=1)  # Evaluate the model  loss, acc = model.evaluate(x\_arrayReshaped, y\_arrayReshaped, verbose=0)  print('Test Accuracy: %.3f' % acc)  # Make a prediction  row = [4.5,1]  yhat = model.predict([row])  print('Predicted: %.3f' % yhat) |

Exercise 5 (1 mark)

Starting with Example 2, replace the code in the first hidden layer with:

|  |
| --- |
| model.add(Dense(3, activation='sigmoid', input\_shape=(2,))) |

Does the code work? Explain in your own words why or why not. How many neurons are in the first hidden layer after the change?

|  |
| --- |
| Yes the code worked. This is because the input\_shape is still accurate to the requirements of 2 feature columns. There are 3 neurons in the first hidden layer after the change. |

Exercise 6 (1 mark)

Starting with Example 2, replace the code in the first hidden layer with:

|  |
| --- |
| model.add(Dense(10, activation='sigmoid', input\_shape=(4,))) |

Does the code work? Explain in your own words why or why not.

|  |
| --- |
| No the code does not work because the input shape is not the dimensions we need in this case. |

Exercise 7 (1 mark)

The loss function option, sparse\_categorical\_crossentropy, is used to generate classification predictions when the target variable is expressed as an non-binary integer. For example, a target variable that qualifies could have the values: 0, 1, 2. Starting with Example 2, replace the compile instruction with:

|  |
| --- |
| model.compile(loss='sparse\_categorical\_crossentropy', metrics=['accuracy']) |

Does the code work? Explain in your own words why or why not.

|  |
| --- |
| No, the code does not work. I believe this is because we are attempting to use an incorrect category of loss function for the binary predictions. |

Exercise 8 (1 mark)

If you remove the second hidden layer from Example 2, does the code work? What conclusion can you make based on the result of removing the second hidden layer?

|  |
| --- |
| Yes, the code still works after removing the second hidden layer. The conclusion that I made was that hidden layers can be added and removed to improve performance or accuracy. |

Exercise 9 (8 marks)

In this exercise you will build a neural network to make predictions for the Iris data set. Please use the iris\_old.csv file that is in the data sets folder within COMP4948. This first block of code helps to prepare DataFrame objects to store the X and Y values.

|  |
| --- |
| import pandas as pd  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import LabelEncoder  from tensorflow.keras import Sequential  from tensorflow.keras.layers import Dense  PATH = "/Users/pm/Desktop/DayDocs/data/"  df = pd.read\_csv(PATH + 'iris\_old.csv')  df.columns = ['Sepal L', 'Sepal W', 'Petal L', 'Petal W', 'Iris Type']  print(df)  # Convert text to numeric category.  # 0 is setosa, 1 is versacolor and 2 is virginica  df['y'] = LabelEncoder().fit\_transform(df['Iris Type'])  # Prepare the data.  dfX = df.iloc[:, 0:4] # Get X features only from columns 0 to 3  dfY = df.iloc[:, 5:6] # Get X features only from column 5 |

Your next task is to resize the data into vertical arrays. See Example 2 for a demonstration of how to do this. Be sure to read the discussion which leads up to the code for additional information on how to perform this task. Next, after reshaping your data, use the following code to split it into test and training sets.

|  |
| --- |
| # Split into train and test data sets.  X\_train, X\_test, y\_train, y\_test = train\_test\_split(  x\_arrayReshaped, y\_arrayReshaped, test\_size=0.33) |

After, address these requirements:

* create a 12:3 network. In other words, the first hidden layer is 12 neurons and the output layer is 3 neurons.
* Use a ***sigmoid*** activation function for your first hidden layer.
* Use a ***softmax*** activation function for the output layer. Use ***sparse\_categorical\_crossentropy*** as the option for the loss function when compiling the model. (‘softmax’ allows a multi-class output. ‘*sparse\_categorical\_crossentropy’* allows for multi-class integer labels.)
* Evaluate the function with the test data. Your accuracy should be over 90% if you performed the tasks properly.
* Make a prediction with the following data set: row = [5.1, 3.5, 1.4, 0.2]. Display the predicted Iris type for the test row.

Show your full program here:

|  |
| --- |
| *import* pandas                *as*     pd  *from* sklearn.model\_selection *import* train\_test\_split  *from* sklearn.preprocessing   *import* LabelEncoder  *from* tensorflow.keras        *import* Sequential  *from* tensorflow.keras.layers *import* Dense  *from* tensorflow.keras.layers *import* Dense  *import* numpy  *as* np  PATH       = "/Users/pm/Desktop/DayDocs/data/"  df         = pd.read\_csv('iris\_old.csv')  df.columns = ['Sepal L', 'Sepal W', 'Petal L', 'Petal W', 'Iris Type']  print(df)  *# Convert text to numeric category.*  *# 0 is setosa, 1 is versacolor and 2 is virginica*  df['y'] = LabelEncoder().fit\_transform(df['Iris Type'])  *# Prepare the data.*  dfX = df.iloc[:, 0:4] *# Get X features only from columns 0 to 3*  dfY = df.iloc[:, 5:6] *# Get X features only from column 5*  ROW\_DIM = 0  COL\_DIM = 1  x\_array = dfX.values  x\_arrayReshaped = x\_array.reshape(x\_array.shape[ROW\_DIM],                                    x\_array.shape[COL\_DIM])  *# Convert DataFrame columns to vertical columns of target variables values.*  y\_array = dfY.values  y\_arrayReshaped = y\_array.reshape(y\_array.shape[ROW\_DIM],                                    y\_array.shape[COL\_DIM])  *# Build a network model of sequential layers.*  model = Sequential()  *# Add 1st hidden layer. Note 1st hidden layer also receives data from input layer.*  *# The input array must contain two feature columns and any number of rows.*  model.add(Dense(12, *activation*='sigmoid',  *input\_shape*=(x\_arrayReshaped.shape[COL\_DIM],)))  *# Add output layer.*  model.add(Dense(3,  *activation*='softmax'))  *# Compile the model.*  *# Binary cross entropy is used to measure error cost for binary predictions.*  model.compile(*loss*='sparse\_categorical\_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\_arrayReshaped, y\_arrayReshaped, *epochs*=1000, *verbose*=1)  *# Evaluate the model*  loss, acc = model.evaluate(x\_arrayReshaped, y\_arrayReshaped, *verbose*=0)  print('Test Accuracy: %.3f' % acc)  *# Make a prediction*  row = [5.1, 3.5, 1.4, 0.2]  yhat = model.predict([row])  print(f'Predicted: {yhat}') |