**Iris modelling using Feed Forward Neural Network**

Fisher’s Iris database (Fisher, 1936) is perhaps the best known database to be found in the pattern recognition literature. The data set contains 3 classes of 50 instances(Samples) each, where each class refers to a type of iris plant. One class is linearly separable from the other two; the latter are not linearly separable from each other. The database contains the following attributes:

1). sepal length in cm

2). sepal width in cm

3). petal length in cm

4). petal width in cm

5). class:

- Iris Setosa

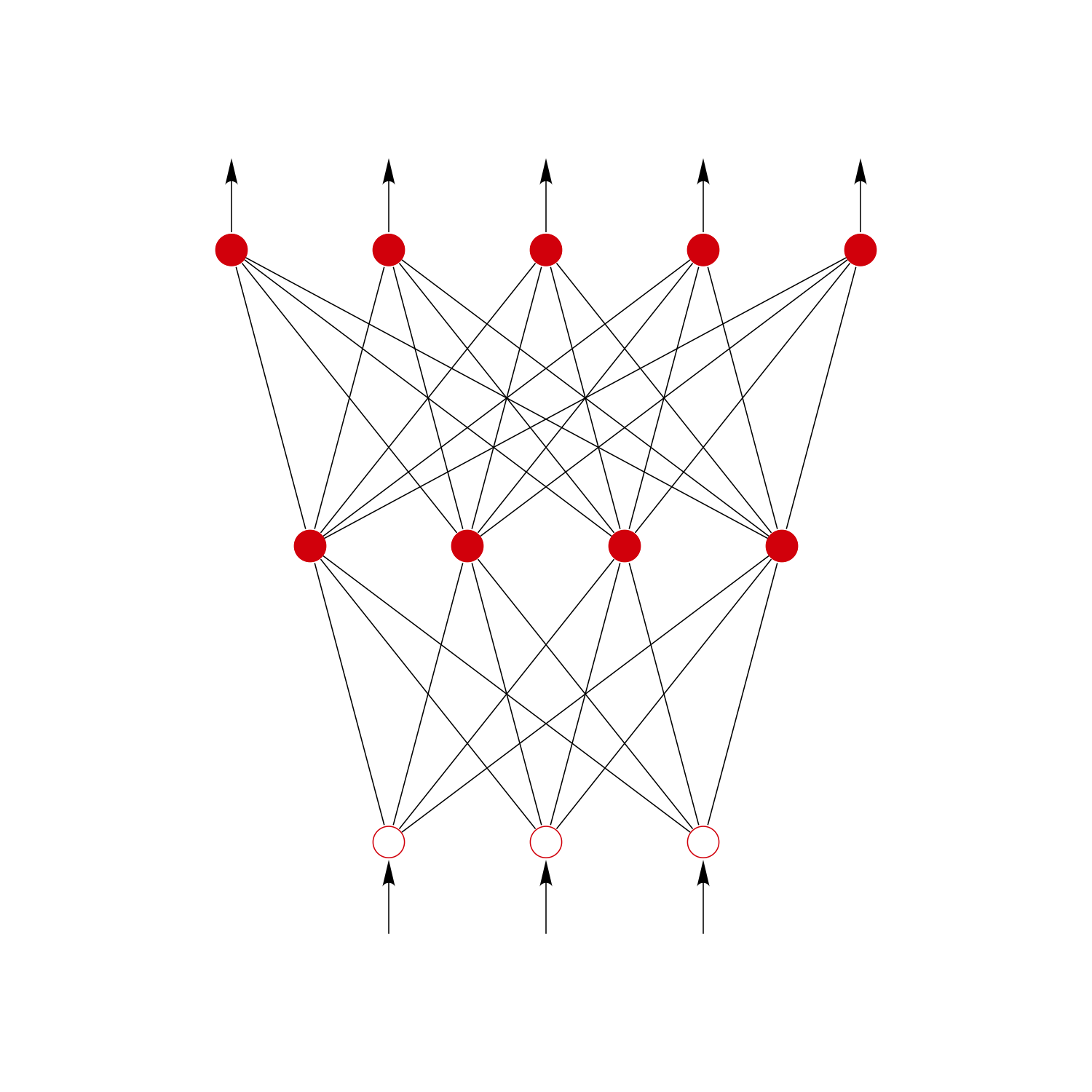
- Iris Versicolour

- Iris Virginica

A feedforward neural network is a biologically inspired classification algorithm. It consists of a (possibly large) number of simple neuron-like processing units, organized in layers. Every unit in a layer is connected with all the units in the previous layer. These connections are not all equal: each connection may have a different strength or weight. The weights on these connections encode the knowledge of a network. Often the units in a neural network are also called nodes.

Data enters at the inputs and passes through the network, layer by layer, until it arrives at the outputs. During normal operation, that is when it acts as a classifier, there is no feedback between layers. This is why they are called feedforward neural networks.

In the following figure we see an example of a 2-layered network with, from top to bottom: an output layer with 5 units, a hidden layer with 4 units, respectively. The network has 3 input units.



The 3 inputs are shown as circles and these do not belong to any layer of the network (although the inputs sometimes are considered as a virtual layer with layer number 0). Any layer that is not an output layer is a hidden layer. This network therefore has 1 hidden layer and 1 output layer. The figure also shows all the connections between the units in different layers. A layer only connects to the previous layer.

Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range -1 to +1.

**How to load iris.csv dataset**

Load CSV files to Python Pandas

1. Load the Pandas libraries with alias 'pd'
2. import pandas as pd.
3. Read data from file 'filename.csv'
4. (in the same directory that your python process is based)
5. Control delimiters, rows, column names with read\_csv
6. data = pd.read\_csv ( url or path of iris.csv )

The following code is to download iris dataset

|  |
| --- |
| import pandas as pd data = pd.read\_csv('iris/Iris.csv')  print(data) |

**<compiler button>**

**Split the iris dataset into train set and test set**

train\_test\_split is a function in Sklearn model selection for splitting data arrays into two subsets: for training data and for testing data. With this function, you don't need to divide the dataset manually. By default, Sklearn train\_test\_split will make random partitions for the two subsets.

We are going to separate the data. One part will be used to make predictions in the end, the other part, the most important will be used for training and testing the neural network.

|  |
| --- |
| data = data.drop(['Id'], axis =1) X = data.drop(['Species'], axis = 1) Y = data['Species'] |

**<compiler button>**

We must transform the column of classes, because we have a format 'str', and it is a multiclass situation. We must first convert the names of species into numerical values, then into vectors for the output of the neuron network.

|  |
| --- |
| Y=pd.get\_dummies(Y) import sklearn  from sklearn import preprocessing, model\_selection  train\_x, test\_x, train\_y, test\_y = model\_selection.train\_test\_split(X,Y,test\_size = 0.1, random\_state = 0) |

**<compiler button>**

1. test\_size — This parameter decides the size of the data that has to be split as the test dataset. ...
2. train\_size — You have to specify this parameter only if you're not specifying the test\_size. ...
3. random\_state — Here you pass an integer, which will act as the seed for the random number generator during the split.

**Creating a sequential model using keras**

The simplest model is defined in the Sequential class which is a linear stack of Layers. You can create a Sequential model and define all of the layers in the constructor, for example:

|  |
| --- |
| **from** keras.models **import** Sequential  model = Sequential(...) |

From keras the model is building

(or)

|  |
| --- |
| **import** tensorflow **as** tf model = tf.keras.Sequential() |

From tensorflow the model is building

You can choose any one from these. But here we are choosing tensorflow

The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that share layers or have multiple inputs or outputs.

The Sequential model API is a way of creating deep learning models where an instance of the Sequential class is created and model layers are created and added to it.

|  |
| --- |
| import tensorflow as tf  input\_dim = len(data.columns) - 1  print(input\_dim)  model = tf.keras.Sequential()  OUTPUT:  4 |

**<compiler button>**

The model type that we will be using is Sequential. Sequential is the easiest way to build a model in Keras(or) tensorflow . It allows you to build a model layer by layer. Each layer has weights that correspond to the layer that follows it.

**Adding number of layers**

A dense layer is just a regular layer of neurons in a neural network. Each neuron receives input from all the neurons in the previous layer, thus densely connected. The layer has a weight matrix W, a bias vector b, and the activations of previous layer a.

|  |
| --- |
| model.add(tf.keras.layers.Dense(8, input\_dim = input\_dim , activation = 'relu'))  model.add(tf.keras.layers.Dense(10, activation = 'relu'))  model.add(tf.keras.layers.Dense(10, activation = 'relu'))  model.add(tf.keras.layers.Dense(10, activation = 'relu'))  model.add(tf.keras.layers.Dense(3, activation = 'softmax'))  model.summary() |

**<compiler button>**

**Output:**

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense (Dense) (None, 8) 40

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 90

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 10) 110

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_3 (Dense) (None, 10) 110

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_4 (Dense) (None, 3) 33

=================================================================

Total params: 383

Trainable params: 383

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Notice how the first Dense object specified in the array is not the input layer. The first Dense object is the first hidden layer. The input layer is specified as a parameter to the first Dense object’s constructor.

This is why our input dimensions are specified as input\_dim= input\_dim . Our first hidden layer has eight nodes and the second hidden layer has 10 and does upto four layers, and the fifth layer has three nodes.

For now, just note that we are using an activation function called relu activation='relu' for both of our hidden layers and an activation function called softmax activation='softmax' for our output layer.

We use the ‘add()’ function to add layers to our model.

‘Dense’ is the layer type. Dense is a standard layer type that works for most cases. In a dense layer, all nodes in the previous layer connect to the nodes in the current layer.

**Compiling the model with respective loss functions and optimizers**

It is time to make predictions with the small sample removed from the base at the beginning. To train the neural network it was necessary to convert the species into vectors. So after the prediction it is necessary to carry out the opposite operation to recover the name of the associated species.

Neural networks are trained using an optimization process that requires a loss function to calculate the model error. Cross-entropy and mean squared error are the two main types of loss functions to use when training neural network models.

The purpose of loss functions is to compute the quantity that a model should seek to minimize during training.

|  |
| --- |
| model.compile(  loss=tf.keras.losses.BinaryCrossentropy(from\_logits=True),  optimizer='adam',  metrics=['accuracy']) |

**<compiler button>**

Loss is nothing but a prediction error of Neural Net. And the method to calculate the loss is called Loss Function. In simple words, the Loss is used to calculate the gradients. And gradients are used to update the weights of the Neural Net.

Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses. Optimizers help to get results faster.

Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses. Optimization algorithms or strategies are responsible for reducing the losses and to provide the most accurate results possible.

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data. The algorithm is called Adam. It is not an acronym and is not written as “ADAM”. The name Adam is derived from adaptive moment estimation.

A metric is a function that is used to judge the performance of your model. Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. Note that you may use any loss functions as a metric function.

**Training the model**

Now we will train our model. To train, we will use the ‘fit()’ function on our model with the following four parameters: training data (train\_x), target data (train\_y), the number of epochs and batch\_size.

|  |
| --- |
| model.fit(train\_x, train\_y, epochs = 10, batch\_size = 2) |

**<compiler button>**

**Output:**

Train on 135 samples

Epoch 1/10

135/135 [==============================] - 0s 1ms/sample - loss: 0.5854 - accuracy: 0.9605

Epoch 2/10

135/135 [==============================] - 0s 1ms/sample - loss: 0.5814 - accuracy: 0.9802

Epoch 3/10

135/135 [==============================] - 0s 1ms/sample - loss: 0.5819 - accuracy: 0.9753

Epoch 4/10

135/135 [==============================] - 0s 1ms/sample - loss: 0.5796 - accuracy: 0.9802

Epoch 5/10

135/135 [==============================] - 0s 2ms/sample - loss: 0.5798 - accuracy: 0.9753

Epoch 6/10

135/135 [==============================] - 0s 2ms/sample - loss: 0.5808 - accuracy: 0.9802

Epoch 7/10

135/135 [==============================] - 0s 1ms/sample - loss: 0.5820 - accuracy: 0.9753

Epoch 8/10

135/135 [==============================] - 0s 2ms/sample - loss: 0.5790 - accuracy: 0.9802

Epoch 9/10

135/135 [==============================] - 0s 1ms/sample - loss: 0.5805 - accuracy: 0.9753

Epoch 10/10

135/135 [==============================] - 0s 2ms/sample - loss: 0.5820 - accuracy: 0.9704

<tensorflow.python.keras.callbacks.History at 0x7ff0c8498910>

The number of epochs is the number of times the model will cycle through the data. The more epochs we run, the more the model will improve, up to a certain point.

You should set the number of epochs as high as possible and terminate training based on the error rates. Just to be clear, an epoch is one learning cycle where the learner sees the whole training data set. If you have two batches, the learner needs to go through two iterations for one epoch.

The batch size is a number of samples processed before the model is updated. The number of epochs is the number of complete passes through the training dataset. The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset.

In the neural network terminology: one epoch = one forward pass and one backward pass of all the training examples. batch size = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need

**Evaluating the models accuracy**

Model Evaluation is an integral part of the model development process. It helps to find the best model that represents our data and how well the chosen model will work in the future. To avoid overfitting, both methods use a test set (not seen by the model) to evaluate model performance.

|  |
| --- |
| scores = model.evaluate(test\_x, test\_y) print("Test score",scores[0]) print("Test accuracy",scores[1]) |

**<compiler button>**

**Output:**

15/15 [==============================] - 0s 174us/sample - loss: 0.5799 - accuracy: 0.9556

('Test score', 0.5798901319503784)

('Test accuracy', 0.95555556)

The model. evaluate function predicts the output for the given input and then computes the metrics function specified in the model. So even if you use the same data, the differences will be there because the value of a loss function will be almost always different than the predicted values.