Artificial Neural Networks and The Multi-Layer Perceptron

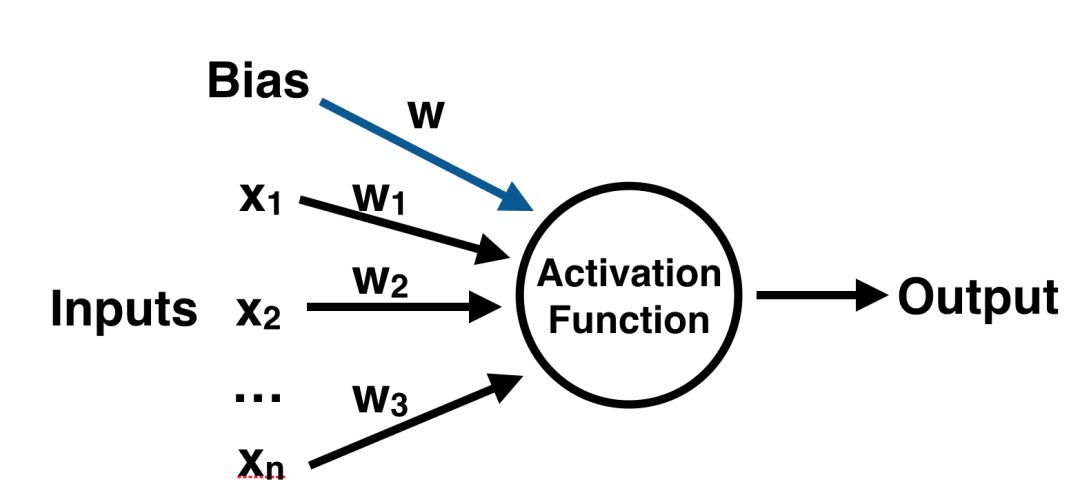
The process of creating a neural network begins with the *perceptron*. In simple terms, the perceptron receives inputs, multiplies them by some weights, and then passes them into an activation function (such as logistic, relu, tanh, identity) to produce an output.

Neural networks are created by adding the layers of these perceptrons together, known as a **multi-layer perceptron** model. There are three layers of a neural network - the input, hidden, and output layers. The *input layer* directly receives the data, whereas the *output layer* creates the required output. The layers in between are known as *hidden layers* where the intermediate computation takes place.

A neural network algorithm can be used for both classification and regression problems. Before we start building the model, we will gain an understanding of the problem statement and the data.

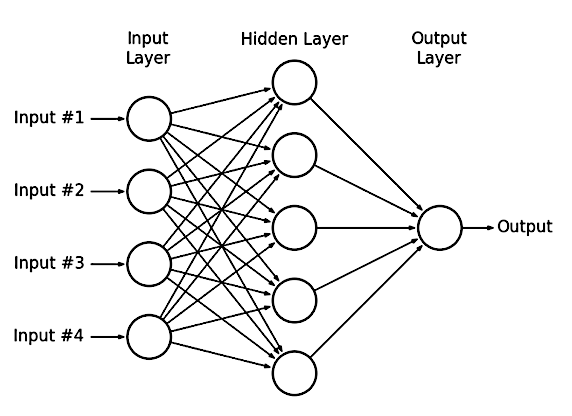
## The McCulloch-Pitts neuron (The Perceptron)

Let’s start our discussion by talking about the Perceptron! A perceptron has one or more inputs, a bias, an activation function, and a single output. The perceptron receives inputs, multiplies them by some weight, and then passes them into an activation function to produce an output. There are many possible activation functions to choose from, such as the logistic function, a trigonometric function, a step function etc. We must also make sure to add a bias to the perceptron, a constant weight outside of the inputs that allows us to achieve better fit for our predictive models. Check out the diagram below for a visualization of a perceptron:



Once we have the output, we can compare it to a known label and adjust the weights accordingly (the weights usually start off with random initialization values) with backpropagation. We keep repeating this process until we have reached a maximum number of allowed iterations, or an acceptable error rate.

To create a neural network, we simply begin to add layers of perceptrons together, creating a multi-layer perceptron model of a neural network. You’ll have an input layer which directly takes in your data and an output layer which will create the resulting outputs. Any layers in between are known as hidden layers because they don’t directly “see” the feature inputs within the data you feed in or the outputs. For a visualization of this check out the diagram below.



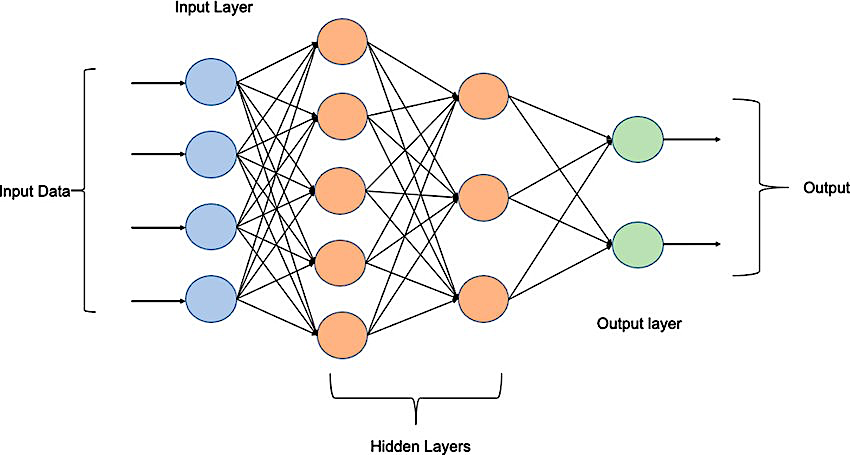
Keep in mind that due to their nature, neural networks tend to work better on GPUs than on CPU. The sci-kit learn framework isn’t built for GPU optimization. If you want to continue using GPUs and distributed models, take a look at some other frameworks, such as Google’s open sourced [TensorFlow](https://github.com/tensorflow/tensorflow).

Multi-Layer Perceptron Learning

Multi-Layer Perceptron (MLP) is an artificial neural network widely used for solving classification and regression tasks. MLP consists of fully connected dense layers that transform input data from one dimension to another. It is called *“multi-layer”* because it contains an input layer, one or more hidden layers, and an output layer. The purpose of an MLP is to model complex relationships between inputs and outputs, making it a powerful tool for various machine learning tasks.

Key Components of Multi-Layer Perceptron (MLP)

* **Input Layer**: Each neuron (or node) in this layer corresponds to an input feature. For instance, if you have three input features, the input layer will have three neurons.
* **Hidden Layers**: An MLP can have any number of hidden layers, with each layer containing any number of nodes. These layers process the information received from the input layer.
* **Output Layer**: The output layer generates the final prediction or result. If there are multiple outputs, the output layer will have a corresponding number of neurons.



Every connection in the diagram is a representation of the fully connected nature of an MLP. This means that every node in one layer connects to every node in the next layer. As the data moves through the network, each layer transforms it until the final output is generated in the output layer.

Working of Multi-Layer Perceptron

Let’s delve in to the working of the multi-layer perceptron. The key mechanisms such as forward propagation, loss function, backpropagation, and optimization.

**Forward Propagation (Predictions)**

In forward propagation, the data flows from the input layer to the output layer, passing through any hidden layers. Each neuron in the hidden layers processes the input as follows:

1. **Weighted Sum:** The neuron computes the weighted sum of the inputs:
   * + Where:
       - is the input feature.
       - is the corresponding weight.
       - is the bias term.
2. **Activation Function:** The weighted sum z is passed through an activation function to introduce non-linearity. Common activation functions include:
   * **Sigmoid:**
   * **ReLU (Rectified Linear Unit):**
   * **Tanh (Hyperbolic Tangent):**
3. **Loss Function:** Once the network generates an output, the next step is to calculate the loss using a loss function. In supervised learning, this compares the predicted output to the actual label.

For a classification problem, the commonly used **binary cross-entropy loss function** is:

* + - Where:
      * is the actual label.
      * is the predicted label
      * is the number of samples

For regression problems, the **mean squared error (MSE)** is often used:

1. **Backpropagation:** The goal of training an MLP is to minimize the loss function by adjusting the network’s weights and biases. This is achieved through **backpropagation**:

* **Gradient Calculation**: The gradients of the loss function with respect to each weight and bias are calculated using the chain rule of calculus.
* **Error Propagation**: The error is propagated back through the network, layer by layer.
* **Gradient Descent**: The network updates the weights and biases by moving in the opposite direction of the gradient to reduce the loss:
* is the weight
* is the learning rate
* is the gradient of the loss function with respect to the weight

1. **Optimisation (solver):** MLPs rely on optimisation algorithms to iteratively refine the weights and biases during training. Popular optimisation methods include:

* **Stochastic Gradient** Descent (SGD): Updates the weights based on a single sample or a small batch of data.
* **Adam Optimizer:** An extension of SGD that incorporates momentum and adaptive learning rates for more efficient training.

1. **Hidden Layer Sizes:** A hyperparameter, accepts tuple of integer specifying sizes of hidden layers in multi-layer perceptrons. According to size of tuple, that many perceptrons will be created per hidden layer. default=(100,)

House Prices Prediction (Regression) with MLP

This dataset contains information collected by the U.S Census Service concerning housing in the area of Boston Mass. It was obtained from the StatLib archive (<http://lib.stat.cmu.edu/datasets/boston>), and has been used extensively throughout the literature to benchmark algorithms. However, these comparisons were primarily done outside of **Delve** and are thus somewhat suspect.

The data was originally published by Harrison, D. and Rubinfeld, D.L. `*Hedonic prices and the demand for clean air*', J. Environ. Economics & Management, vol.5, 81-102, 1978.

**1) Loading the libraries**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.neural\_network import MLPClassifier

from sklearn.neural\_network import MLPRegressor

from sklearn import metrics

from sklearn.model\_selection import GridSearchCV

**2) Loading the dataset**

# To expand e scientific notation

pd.set\_option('display.float\_format', '{:.2f}'.format)

df = pd.read\_csv('/content/house\_prices.csv')

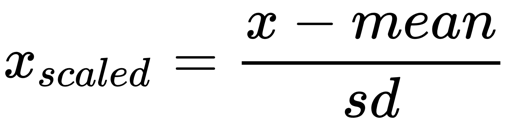
df.describe(include="all").transpose()

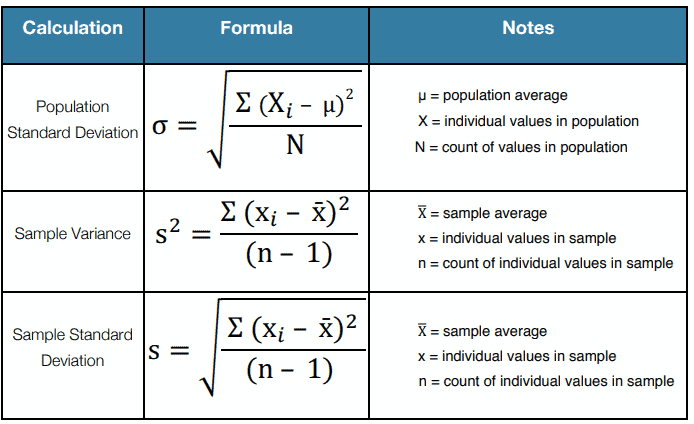
**A screenshot of a graph

Description automatically generated**

**3) Preprocess your dataset**

The neural network in Python may have difficulty converging before the maximum number of iterations allowed if the data is not normalised. Multi-layer Perceptron is sensitive to feature scaling, so it is highly recommended to scale your data. Note that you must apply the same scaling to the test set for meaningful results. There are a lot of different methods for normalisation of data; we will use the built-in **StandardScaler** for standardisation.





#Assign your input and output features

x = df.drop(['price', 'id'], axis=1)

y = df['price']

#Split your data into training and testing

trainX, testX, trainY, testY = train\_test\_split(x, y, test\_size = 0.5)

# Scale your dataset with standard scaler

sc=StandardScaler()

scaler = sc.fit(trainX)

trainX\_scaled = scaler.transform(trainX)

testX\_scaled = scaler.transform(testX)

**4) Initiate and fit the MLP Regressor:** Here, you define a range of hyperparameters, including the activation function type, the number of hidden layers and the number of neurons within each layer. For backpropagation, we need to set the number of iterations required to adjust the wights and biases (training). We also choose a solver algorithm to optimise the weights, in this case we use “Adam”.

mlp\_reg = MLPRegressor(hidden\_layer\_sizes=(150,100,50), max\_iter = 300,activation = 'relu', solver = 'adam')

mlp\_reg.fit(trainX\_scaled, trainY)

**A screenshot of a computer

Description automatically generated**

**5) Evaluating your regression model:** perform predictions on the test dataset then calculate the regression metrics i.e. MAE, MSE, R2 and RMSE.

y\_pred = mlp\_reg.predict(testX\_scaled)

prediction\_table = pd.DataFrame({'Actual': testY, 'Predicted': y\_pred})

prediction\_table.head(10)

**A screenshot of a graph

Description automatically generated**

print('Mean Absolute Error:', metrics.mean\_absolute\_error(testY, y\_pred))

print('Mean Absolute Percentage Error MAPE:', metrics.mean\_absolute\_percentage\_error(testY, y\_pred))

print('Mean Squared Error:', metrics.mean\_squared\_error(testY, y\_pred))

print('Root Mean Squared Error:', np.sqrt(metrics.mean\_squared\_error(testY, y\_pred)))

print('Mean R-Square:', metrics.r2\_score(testY, y\_pred))

**A computer error message on a black background

Description automatically generated**

Wine Fraud Prediction (Classification) with MLP

For this analysis, we will cover one of life’s most important topics – Wine fraud! All joking aside, [wine fraud](https://en.wikipedia.org/wiki/Wine_fraud) is a very real thing. Let’s see if a Neural Network in Python can help with this problem! We will use the wine data set from the UCI Machine Learning Repository. It has various chemical features of different wines, all grown in the same region in Italy, but the data is labelled by three different possible cultivars. We will try to build a model that can classify what cultivar a wine belongs to based on its chemical features using Neural Networks. You can get the data [here](https://archive.ics.uci.edu/ml/datasets/Wine) or find other free [data sets here](https://www.springboard.com/blog/free-public-data-sets-data-science-project/).

First, let’s import the dataset! We’ll use the names feature of Pandas to make sure that the column names associated with the data come through. These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines **{1: Excellent, 2: Normal, 3: Poor}**.There are many more normal wines than excellent or poor ones.

A screenshot of a cell phone

Description automatically generated

**1) Loading the dataset and examining the data types and the summary statistics**

df = pd.read\_csv('/content/wine.csv')

df.info()

A screenshot of a computer

Description automatically generated

print(df.shape)

df.describe().transpose()

A screenshot of a graph

Description automatically generated

#Check the distinct number of classes in the target variable

df['Wine'].unique()



**2) Preprocess your dataset**

The neural network in Python may have difficulty converging before the maximum number of iterations allowed if the data is not normalised. Multi-layer Perceptron is sensitive to feature scaling, so it is highly recommended to scale your data. Note that you must apply the same scaling to the test set for meaningful results. There are a lot of different methods for normalisation of data; we will use the built-in **StandardScaler** for standardisation. First, we define the input and output variables and split the dataset into training and test subsets. Finally, the standard scaler will be applied to both training and test input data.

#Let’s define our inputs and target variables then perform the training test split

X = df.drop('Wine',axis=1)

y = df['Wine']

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y)

#Scale the Train and Test Data

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# Fit only to the training data

scaler.fit(X\_train)

# Now apply the transformations to the data:

X\_train = scaler.transform(X\_train)

X\_test = scaler.transform(X\_test)

**3) Apply GridSearchCV to tune your MLP:** We can calculate the best parameters for the model using “GridSearchCV”. The input parameters for the GridSearchCV method are:

1. The MLP model

2. A parameter dictionary in which we define various hidden layers, activation units, learning rates. it trains the model and finds the best parameter.

%%time

from sklearn.model\_selection import GridSearchCV

params = {'activation': ['relu', 'tanh', 'logistic', 'identity'],

'hidden\_layer\_sizes': [(13,), (50,100,), (50,75,100,)],

'solver': ['adam', 'sgd', 'lbfgs'],

'learning\_rate' : ['constant', 'adaptive', 'invscaling'],

'max\_iter': [500]

}

mlp\_classif\_grid = GridSearchCV(MLPClassifier(random\_state=123), param\_grid=params, n\_jobs=-1, cv=5, verbose=5)

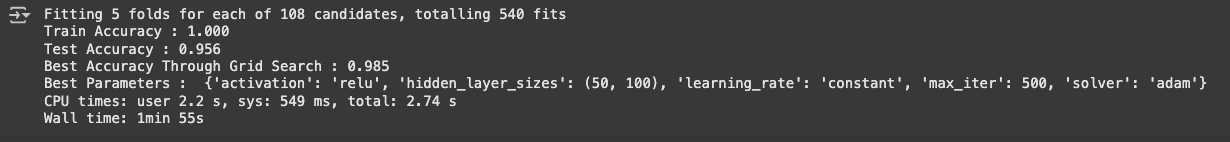
mlp\_classif\_grid.fit(X\_train,y\_train)

print('Train Accuracy : %.3f'%mlp\_classif\_grid.best\_estimator\_.score(X\_train, y\_train))

print('Test Accuracy : %.3f'%mlp\_classif\_grid.best\_estimator\_.score(X\_test, y\_test))

print('Best Accuracy Through Grid Search : %.3f'%mlp\_classif\_grid.best\_score\_)

print('Best Parameters : ',mlp\_classif\_grid.best\_params\_)

****

Now that the model has been made, we can fit the training data to our model, remember that this data has already been processed and scaled:

mlp = MLPClassifier(activation= 'relu', hidden\_layer\_sizes= (50,100), learning\_rate='constant', solver='adam', max\_iter=500)

mlp.fit(X\_train,y\_train)

**A blue and white box with black text

Description automatically generated**

**4) Predictions and Performance Evaluation:** Now that we have a model it is time to use it to get predictions! We can do this simply with the **predict()** method to our fitted model on the test data subset. Then, use SciKit-Learn’s built in metrics such as a classification report and confusion matrix to evaluate how well our model performed:

#Predict the class values for the test data, thus these are called predictions

predictions = mlp.predict(X\_test)

# import all the required packages for classification model evaluation

from sklearn.metrics import classification\_report,confusion\_matrix, ConfusionMatrixDisplay

# Construct and plot the confusion matrix for the fuly-grown tree (clf model)

mlp\_cm\_test = confusion\_matrix(y\_test, predictions, labels=mlp.classes\_ )

disp = ConfusionMatrixDisplay(mlp\_cm\_test,display\_labels=mlp.classes\_ )

disp.plot()

**A chart with a yellow square and a purple square

Description automatically generated**

# display the classification report

print(classification\_report(y\_test,predictions))

A screenshot of a computer program

Description automatically generated