**Neural Networks**

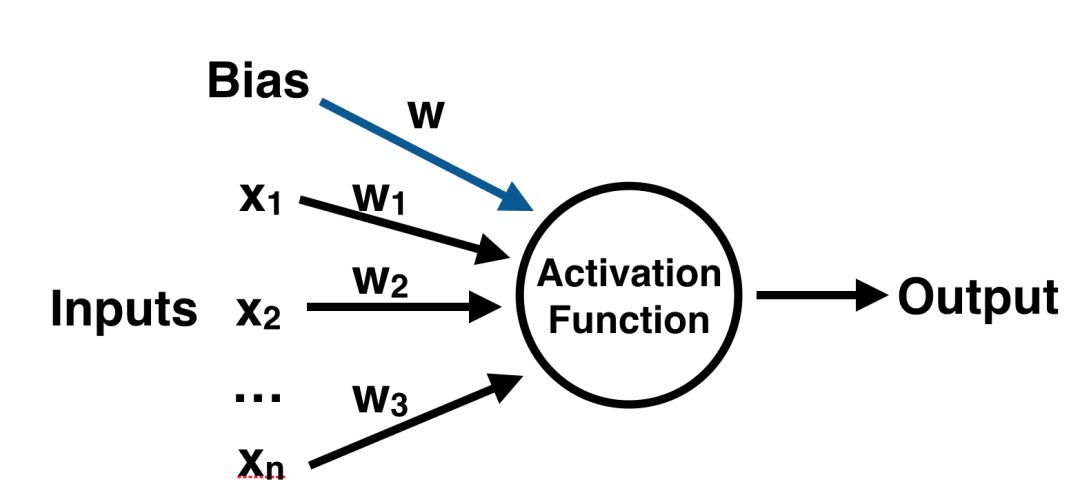
The process of creating a neural network begins with the *perceptron*. In simple terms, the perceptron receives inputs, multiplies them by some weights, and then passes them into an activation function (such as logistic, relu, tanh, identity) to produce an output.

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.

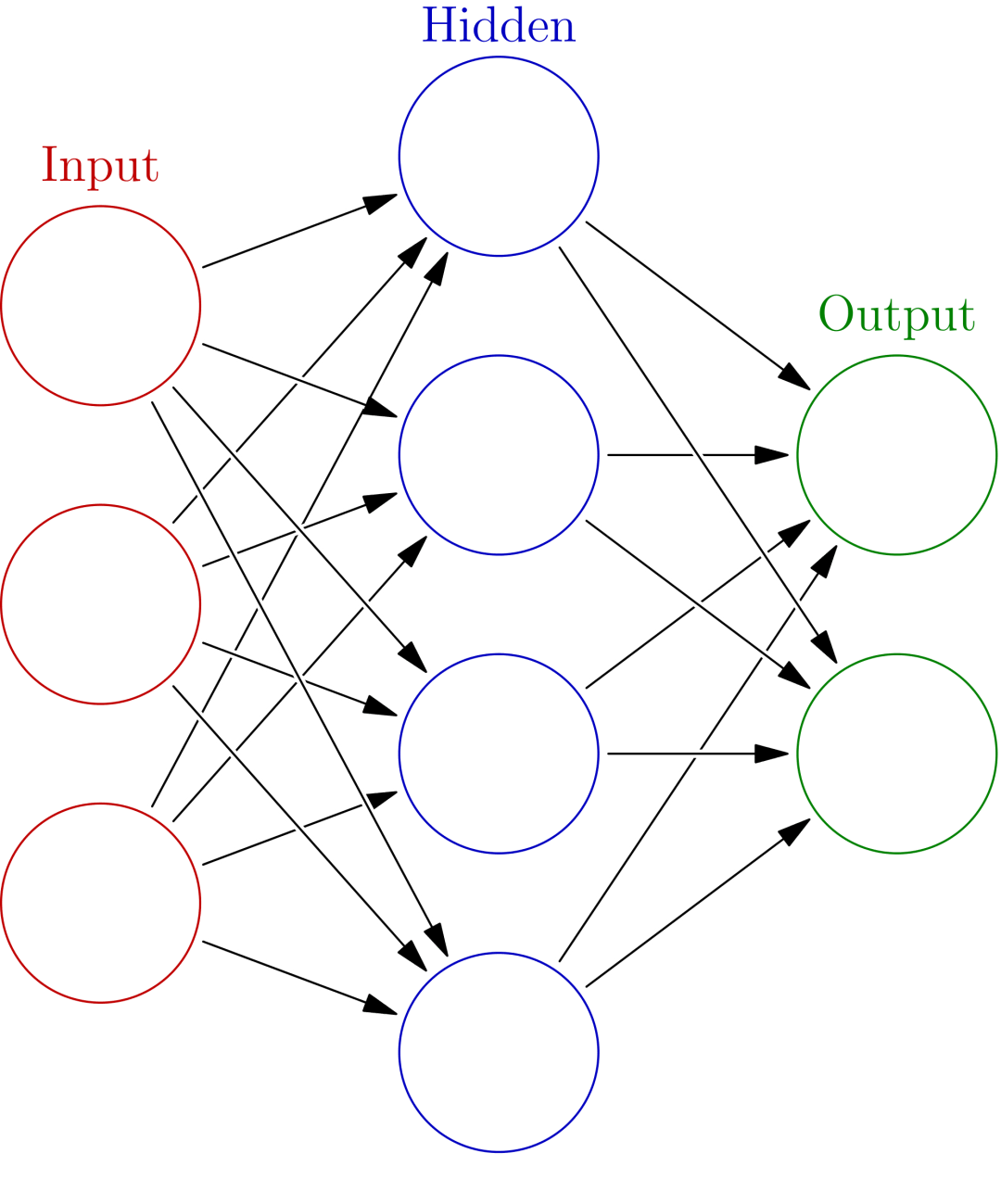
## The Perceptron

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). 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).

**Problem Statement**

The aim of this guide is to build a classification model to detect diabetes. We will be using the diabetes dataset which contains 768 observations and 9 variables, as described below:

1. pregnancies - Number of times pregnant.
2. glucose - Plasma glucose concentration.
3. diastolic - Diastolic blood pressure (mm Hg).
4. triceps - Skinfold thickness (mm).
5. insulin - Hour serum insulin (mu U/ml).
6. bmi – Basal metabolic rate (weight in kg/height in m).
7. dpf - Diabetes pedigree function.
8. age - Age in years.
9. outcome - “1” represents the presence of diabetes while “0” represents the absence of it. This is the target variable.

**Evaluation Metric**

We will evaluate the performance of the model using accuracy, which represents the percentage of cases correctly classified.

Mathematically, for a binary classifier, it's represented as accuracy = (TP+TN)/(TP+TN+FP+FN), where:

* True Positive, or TP, are cases with positive labels which have been correctly classified as positive.
* True Negative, or TN, are cases with negative labels which have been correctly classified as negative.
* False Positive, or FP, are cases with negative labels which have been incorrectly classified as positive.
* False Negative, or FN, are cases with positive labels which have been incorrectly classified as negative.

**Steps**

In this guide, we will follow the following steps:

*Step 1 - Loading the required libraries and modules.*

*Step 2 - Reading the data and performing basic data checks.*

*Step 3 - Creating arrays for the features and the response variable.*

*Step 4 - Creating the training and test datasets.*

*Step 5 - Building , predicting, and evaluating the neural network model.*

The following sections will cover these steps.

**Step 1 - Loading the Required Libraries and Modules**

# Import required libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import sklearn

from sklearn.neural\_network import MLPClassifier

from sklearn.neural\_network import MLPRegressor

# Import necessary modules

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

from sklearn.metrics import mean\_squared\_error

from math import sqrt

from sklearn.metrics import r2\_score

**Step 2 - Reading the Data and Performing Basic Data Checks**

The *first line of code* reads in the data as pandas dataframe, while the *second line* prints the shape - 768 observations of 9 variables. The *third line* gives the transposed summary statistics of the variables.

Looking at the summary for the 'outcome' variable, we observe that the mean value is 0.35, which means that around 35 percent of the observations in the dataset have diabetes. Therefore, the baseline accuracy is 65 percent and our neural network model should definitely beat this baseline benchmark.

df = pd.read\_csv('diabetes.csv')

print(df.shape)

df.describe().transpose()

**Step 3 - Creating Arrays for the Features and the Response Variable**

The *first line of code* creates an object of the target variable called 'target\_column'. The *second line* gives us the list of all the features, excluding the target variable 'outcome, while the *third line* normalizes the predictors.

The *fourth line* displays the summary of the normalized data. We can see that all the independent variables have now been scaled between 0 and 1. The target variable remains unchanged.

target\_column = ['outcome']

predictors = list(set(list(df.columns))-set(target\_column))

df[predictors] = df[predictors]/df[predictors].max()

df.describe().transpose()

**Step 4 - Creating the Training and Test Datasets**

The *first couple of lines of code* below create arrays of the independent (X) and dependent (y) variables, respectively. The *third line* splits the data into training and test dataset, and the *fourth line* prints the shape of the training and the test data.

X = df[predictors].values

y = df[target\_column].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30, random\_state=40)

print(X\_train.shape);

print(X\_test.shape)

**Step 5 - Building, Predicting, and Evaluating the Neural Network Model**

In this step, we will build the neural network model using the scikit-learn library's estimator object, 'Multi-Layer Perceptron Classifier'. The *first line of code* (shown below) imports 'MLPClassifier'.

The *second line* instantiates the model with the 'hidden\_layer\_sizes' argument set to three layers, which has the same number of neurons as the count of features in the dataset. We will also select 'relu' as the activation function and 'adam' as the solver for weight optimization. To learn more about 'relu' and 'adam', please refer to the Deep Learning with Keras guides.

The *third line of code* fits the model to the training data, while the *fourth and fifth lines* use the trained model to generate predictions on the training and test dataset, respectively.

from sklearn.neural\_network import MLPClassifier

mlp = MLPClassifier(hidden\_layer\_sizes=(8,8,8), activation='relu', solver='adam', max\_iter=500)

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

predict\_train = mlp.predict(X\_train)

predict\_test = mlp.predict(X\_test)

Once the predictions are generated, we can evaluate the performance of the model. Being a classification algorithm, we will first import the required modules, which is done in the *first line of code* below. The *second and third lines of code* print the confusion matrix and the confusion report results on the training data.

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

print(confusion\_matrix(y\_train,predict\_train))

print(classification\_report(y\_train,predict\_train))

The above output shows the performance of the model on training data. The accuracy and the F1 score is around 0.78 and 0.77, respectively. Ideally, the perfect model will have the value of 1 for both these metrics, but that is next to impossible in real-world scenarios.

The next step is to evaluate the performance of the model on the test data that is done with the lines of code below.

print(confusion\_matrix(y\_test,predict\_test))

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

The above output shows the performance of the model on test data. The accuracy and F1 scores both around 0.75.

**Conclusion**

In this example, you have learned about building a neural network model using scikit-learn. The guide used the diabetes dataset and built a classifier algorithm to predict the detection of diabetes.

Our model is achieving a decent accuracy of 78 percent and 75 percent on training and test data, respectively. We observe that the model accuracy is higher than the baseline accuracy of 66 percent. The model can be further improved by doing cross-validation, feature engineering, or changing the arguments in the neural network estimator.

Note that we have built a classification model in this guide. However, building the regression model also follows the same structure, with a couple of adjustments. The first being that instead of the estimator 'MLPClassifier', we will instantiate the estimator 'MLPRegressor'. The second adjustment is that, instead of using accuracy as the evaluation metric, we will use RMSE or R-squared value for model evaluation.

**Finetuning Model By Doing Grid Search On Various Hyperparameters.**

Below is a list of common hyperparameters that needs tuning for getting the best fit for our data. We'll try various hyperparameters settings to various splits of train/test data to find out best fit which will have almost the same accuracy for both train & test dataset or have quite less difference between accuracy.

* **hidden\_layer\_sizes** - It 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,)
* **activation** - It specifies activation function for hidden layers. It accepts one of below strings as input. default=relu
  + 'identity' - No Activation. f(x) = x
  + 'logistic' - Logistic Sigmoid Function. f(x) = 1 / (1 + exp(-x))
  + 'tanh' - Hyperbolic tangent function. f(x) = tanh(x)
  + 'relu' - Rectified Linear Unit function. f(x) = max(0, x)
* **solver** - It accepts one of below strings specifying which optimization solver to use for updating weights of neural network hidden layer perceptrons. default='adam'
  + 'lbfgs'
  + 'sgd'
  + 'adam'
* **learning\_rate\_init** - It specifies initial learning rate to be used. Based on value of this parameter weights of perceptrons are updated.default=0.001
* **learning\_rate** - It specifies learning rate schedule to be used for training. It accepts one of below strings as value and only applicable when solver='sgd'.
  + 'constant' - Keeps learning rate constant through a learning process which was set in learning\_rate\_init.
  + 'invscaling' - It gradually decreases learning rate. effective\_learning\_rate = learning\_rate\_init / pow(t, power\_t)
  + 'adaptive' - It keeps learning rate constant as long as loss is decreasing or score is improving. If consecutive epochs fails in decreasing loss according to tol parameter and early\_stopping is on, then it divides current learning rate by 5.
* **batch\_size** - It accepts integer value specifying size of batch to use for dataset. default='auto'. The default auto batch size will set batch size to min(200, n\_samples).
* **tol** - It accepts float values specifying threshold for optimization. When training loss or score is not improved by at least tol for n\_iter\_no\_change iterations, then optimization ends if learning\_rate is constant else it decreases learning rate if learning\_rate is adaptive. default=0.0001
* **alpha** - It specifies L2 penalty coefficient to be applied to perceptrons. default=0.0001
* **momentum** - It specifies momentum to be used for gradient descent and accepts float value between 0-1. It's applicable when solver is sgd.
* **early\_stopping** - It accepts boolean value specifying whether to stop training if training score/loss is not improving. default=False
* **validation\_fraction** - It accepts float value between 0-1 specifying amount of training data to keep aside if early\_stopping is set.default=0.1

## Data

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 labeled 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.

# Import required libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import sklearn

from sklearn.neural\_network import MLPClassifier

from sklearn.neural\_network import MLPRegressor

# Import necessary modules

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

#from sklearn.metrics import mean\_squared\_error

from math import sqrt

#from sklearn.metrics import r2\_score

#Import the Data

df = pd.read\_csv('/Users/user/Desktop/7BUIS008W/wine.csv')

print(df.shape)

df.describe().transpose()

#Let’s set up and split our Data and our Labels

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)

## Data Pre-processing

The neural network in Python may have difficulty converging before the maximum number of iterations allowed if the data is not normalized. 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 normalization of data, we will use the built-in StandardScaler for standardization.

#Scale the Train and Test Data

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# Fit only to the training data

scaler.fit(X\_train)

StandardScaler(copy=True, with\_mean=True, with\_std=True)

# Now apply the transformations to the data:

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

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

## Apply Grid Search 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)

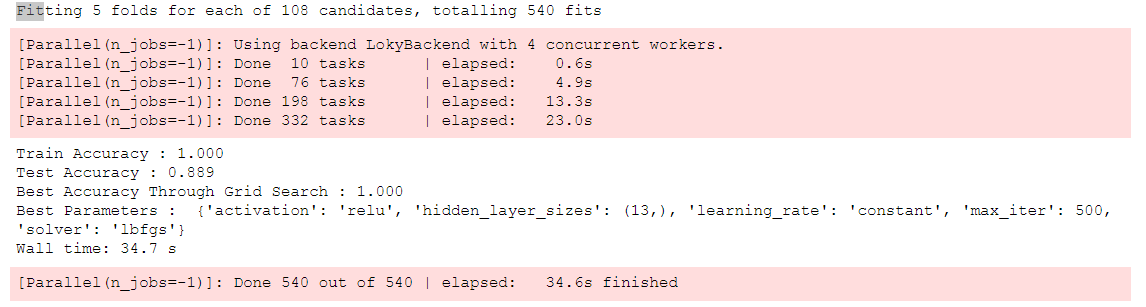
mlp\_classif\_grid.fit(X\_train,y\_train)

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= (13,), learning\_rate='constant', solver='lbfgs', max\_iter=500)

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

## Predictions and Evaluation

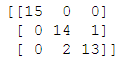
Now that we have a model it is time to use it to get predictions! We can do this simply with the predict() method off of our fitted model:

predictions = mlp.predict(X\_test)

Now we can use SciKit-Learn’s built in metrics such as a classification report and confusion matrix to evaluate how well our model performed:

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

print(confusion\_matrix(y\_test,predictions))



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

