Worksheet for Practical 1

1. Load Data

The first step is to define the functions and classes we intend to use in this tutorial.

We will use the Pandas library to load our dataset and we will use two classes from the Keras library to define our model.

In [16]:
```
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
```

In this worksheet, we are going to use the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years.

As such, it is a binary classification problem (onset of diabetes as 1 or not as 0). All of the input variables that describe each patient are numerical. This makes it easy to use directly with neural networks that expect numerical input and output values, and ideal for our first neural network in Keras.

The variables can be summarized as follows:

Input Variables (X):

Number of times pregnant

Plasma glucose concentration a 2 hours in an oral glucose tolerance test

Diastolic blood pressure (mm Hg)

Triceps skin fold thickness (mm)

2-Hour serum insulin (mu U/ml)

Body mass index (weight in kg/(height in m)^2)

Diabetes pedigree function

Age (years)

Output Variables (y):

Class variable (0 or 1)

Once the CSV file is loaded into memory, we can split the columns of data into input and output variables.

In [17]:
```
# load the dataset
dataset = pd.read_csv('pima-indians-diabetes.csv', delimiter=',')
# split into input (X) and output (y) variables
X = dataset.drop(['Class'], axis = 1)
Y = dataset[['Class']]
```

2. Define Keras Model

Models in Keras are defined as a sequence of layers.

We create a Sequential model (learn about sequential model here: https://www.youtube.com/watch?v=VGCHcgmZu24 (https://www.youtube.com/watch?v=VGCHcgmZu24)) and add layers one at a time until we are happy with our network architecture.

The first thing to get right is to ensure the input layer has the right number of input features. This can be specified when creating the first layer with the input_dim argument and setting it to 8 for the 8 input variables.

In this example, we will use a fully-connected network structure with three layers.

Fully connected layers are defined using the Dense class (learn about dense class here: https://www.youtube.com/watch?v=ohgONsuoxVs (https://www.youtube.com/watch?v=ohgONsuoxVs)). We can specify the number of neurons or nodes in the layer as the first argument, and specify the activation function using the activation argument.

In [18]:
```
# define the keras model
model = Sequential()
model.add(Dense(12, input_dim= 8 , activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

3. Compile Keras Model

Now that the model is defined, we can compile it.

Compiling the model uses the efficient numerical libraries under the covers (the so-called backend) such as Theano or TensorFlow. The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware, such as CPU or GPU or even distributed.

When compiling, we must specify some additional properties required when training the network. Remember training a network means finding the best set of weights to map inputs to outputs in our dataset.

We must specify the loss function to use to evaluate a set of weights, the optimizer is used to search through different weights for the network and any optional metrics we would like to collect and report during training.

In this case, we will use cross entropy as the loss argument. This loss is for a binary classification problems and is defined in Keras as "binary_crossentropy".

We will define the optimizer as the efficient stochastic gradient descent algorithm "adam". This is a popular version of gradient descent because it automatically tunes itself and gives good results in a wide range of problems.

Finally, because it is a classification problem, we will collect and report the classification accuracy, defined via the metrics argument.

In [19]:
```
# compile the keras model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

4. Fit Keras Model

We have defined our model and compiled it ready for efficient computation.

Now it is time to execute the model on some data.

We can train or fit our model on our loaded data by calling the fit() function on the model.

Training occurs over epochs and each epoch is split into batches.

Epoch: One pass through all of the rows in the training dataset.

Batch: One or more samples considered by the model within an epoch before weights are updated.

One epoch is comprised of one or more batches, based on the chosen batch size and the model is fit for many epochs.

The training process will run for a fixed number of iterations through the dataset called epochs, that we must specify using the epochs argument. We must also set the number of dataset rows that are considered before the model weights are updated within each epoch, called the batch size and set using the batch_size argument.

For this problem, we will run for a small number of epochs (150) and use a relatively small batch size of 10.

These configurations can be chosen experimentally by trial and error. We want to train the model enough so that it learns a good (or good enough) mapping of rows of input data to the output classification. The model will always have some error, but the amount of error will level out after some point for a given model configuration. This is called model convergence.

This is where the work happens on your CPU or GPU.

No GPU is required for this worksheet.

In [20]:
```
# fit the keras model on the dataset
model.fit(X, Y, epochs= 150, batch_size=10 )
```

```
Epoch 29/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5535 - accurac
y: 0.7293
Epoch 30/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5638 - accurac
y: 0.7055
Epoch 31/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5703 - accurac
y: 0.7226
Epoch 32/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5861 - accurac
y: 0.7042
Epoch 33/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5535 - accurac
y: 0.7230
Epoch 34/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5766 - accurac
y: 0.7085
Epoch 35/150
77/77 [==============================] - 0s 1ms/step - loss: 0.5733 - accurac
```

5. Evaluate Keras Model

We have trained our neural network on the entire dataset and we can evaluate the performance of the network on the same dataset.

This will only give us an idea of how well we have modeled the dataset (e.g. train accuracy), but no idea of how well the algorithm might perform on new data. We have done this for simplicity, but ideally, you could separate your data into train and test datasets for training and evaluation of your model.

You can evaluate your model on your training dataset using the evaluate() function on your model and pass it the same input and output used to train the model.

This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.

The evaluate() function will return a list with two values. The first will be the loss of the model on the dataset and the second will be the accuracy of the model on the dataset. We are only interested in reporting the accuracy, so we will ignore the loss value.

In [21]:
```
# evaluate the keras model
_, accuracy = model.evaluate(X, Y)
print('Accuracy: %.2f' % (accuracy*100))
```

```
24/24 [==============================] - 0s 1ms/step - loss: 0.4713 - accuracy:
0.7747
Accuracy: 77.47
```