Neural Networks - To predict Whether Patients has diabetics or not .

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation.In this project, 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.It's 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. In this Project we will discover how to create your first neural network model in Python using Keras. 1. How to load a CSV dataset ready for use with Keras. 2. How to define and compile a Multilayer Perceptron model in Keras. 3. How to evaluate a Keras model on a validation dataset.

1. Load Data

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

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

In [4]:
```
#Setup
import warnings
warnings.filterwarnings("ignore")
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, precision_recall_curve, roc_a
from sklearn.ensemble import RandomForestClassifier
```

In [5]:
```
## Import Keras objects for Deep Learning
from keras.models  import Sequential
from keras.layers import Input, Dense, Flatten, Dropout, BatchNormalization
from keras.optimizers import Adam, SGD, RMSprop
```

```
Using TensorFlow backend.
```

In [6]:
```
names = ["times_pregnant", "glucose_tolerance_test", "blood_pressure", "ski
         "bmi", "pedigree_function", "age", "has_diabetes"]
data = pd.read_csv('data/diabetes.csv', names=names, header=0)
```

In [7]:
```
print(data.shape)
```

```
(768, 9)
```

In [8]:
```
#Print no of integers, floats and strings
data.dtypes.value_counts()
```

Out[8]:
```
int64      7
float64    2
dtype: int64
```

In [9]:
```
#Data should be numerical
data.head()
```

Out[9]:

	times_pregnant	glucose_tolerance_test	blood_pressure	skin_thickness	insulin	bmi	pedigree_fu
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

Summary of Input :

There are eight input variables and one output variable (the last column). We will be learning a model to map rows of input variables (X) to an output variable (y), which we often summarize as y = f(X). 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)

In [10]:
```
#Print no of entries
data.has_diabetes.value_counts()
```

Out[10]:
```
0    500
1    268
Name: has_diabetes, dtype: int64
```

In [11]:
```
#Print % of each entries
data.has_diabetes.value_counts(normalize=True)
```

Out[11]:
```
0    0.651042
1    0.348958
Name: has_diabetes, dtype: float64
```

2. Preprocessing Steps

1. Select Features and Split to X and y .

In [12]:
```
X = data.iloc[:, :-1].values
y = data["has_diabetes"].values
```

2. Split data to train and test .

In [13]:
```
# Split the data to Train, and Test (75%, 25%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, r
```

3. Normalize Data

In [14]:
```
## First let's normalize the data
## This aids the training of neural nets by providing numerical stability
normalizer = StandardScaler()
X_train_norm = normalizer.fit_transform(X_train)
X_test_norm = normalizer.transform(X_test)
```

3 .Building a single hidden layer Neural network. (2-layer neural network)¶

1. Define Keras Model

There are two top numerical platforms for developing deep learning models, they are Theano developed by the University of Montreal and TensorFlow developed at Google. Both were developed for use in Python and both can be leveraged by the super simple to use Keras library. Keras wraps the numerical computing complexity of Theano and TensorFlow providing a concise API that we will use to develop our own neural network and deep learning models.

Models in Keras are defined as a sequence of layers. We create a Sequential model and add layers one at a time to our network architecture.

We have input varaibles 8 and we have to set the input to 8. In this Project, we will build a fully-connected network structure with one hidden layer. Fully connected layers are defined using the Dense class. 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. We will use the Sigmoid activation function in layers

The model expects rows of data with 8 variables (the input_shape=8 argument). The one hidden layer has 12 nodes and uses the Sigmoid activation function. The output layer has one node and uses the sigmoid activation function

We use a sigmoid activation function on the output layer to ensure our network output is between 0 and 1 and easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5.

In [15]:
```
# Define the Model
# Input size is 8-dimensional
# 1 hidden layer, 12 hidden nodes, sigmoid activation
# Final layer has just one node with a sigmoid activation (standard for bin

model_1 = Sequential()
model_1.add(Dense(12,input_shape = (8,),activation = 'sigmoid'))
model_1.add(Dense(1,activation='sigmoid'))
```

2. Summary of Keras model and count how many parameters

In [16]:
```
model_1.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 12)                108
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 13
=================================================================
Total params: 121
Trainable params: 121
Non-trainable params: 0
_________________________________________________________________
```

Summary of Prameters to be tuned :

(8 input variables + 1 bias)*12 = 108 12+ 1 bias = 13

3.Compile Kears Model

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 our project, 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 “SGD“.

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

In [17]:
```
# Fit(Train) the Model

# Compile the model with Optimizer, Loss Function and Metrics
# Roc-Auc is not available in Keras as an off the shelf metric yet, so we w

model_1.compile(SGD(lr = .003), "binary_crossentropy", metrics=["accuracy"]
```

4. Fit 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. 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 can also set the number of instances that are evaluated before a weight update in the network is performed called the batch size and set using the batch size argument. For this problem we will run for a small number of epochs 200 . Again, these can be chosen experimentally by trial and error.This is where the work happens on your CPU or GPU.

In [18]:
```
# the fit function returns the run history.
# It is very convenient, as it contains information about the model fit, it
run_hist_1 = model_1.fit(X_train_norm, y_train, validation_data=(X_test_nor
```

```
Train on 576 samples, validate on 192 samples
Epoch 1/200
576/576 [==============================] - 0s 814us/step - loss: 0.6594 -
accuracy: 0.6545 - val_loss: 0.6644 - val_accuracy: 0.6406
Epoch 2/200
576/576 [==============================] - 0s 53us/step - loss: 0.6568 -
accuracy: 0.6545 - val_loss: 0.6616 - val_accuracy: 0.6406
Epoch 3/200
576/576 [==============================] - 0s 54us/step - loss: 0.6543 -
accuracy: 0.6545 - val_loss: 0.6589 - val_accuracy: 0.6406
Epoch 4/200
576/576 [==============================] - 0s 55us/step - loss: 0.6520 -
accuracy: 0.6545 - val_loss: 0.6564 - val_accuracy: 0.6406
Epoch 5/200
576/576 [==============================] - 0s 54us/step - loss: 0.6498 -
accuracy: 0.6545 - val_loss: 0.6540 - val_accuracy: 0.6406
Epoch 6/200
576/576 [==============================] - 0s 53us/step - loss: 0.6477 -
accuracy: 0.6545 - val_loss: 0.6518 - val_accuracy: 0.6406
```

In [21]:
```
## Predicted class and Predicted probability

y_pred_class = model_1.predict_classes(X_test_norm)
y_pred_prob = model_1.predict(X_test_norm)
```

In [22]:
```
y_pred_class[:10]
```

Out[22]:
```
array([[0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0]], dtype=int32)
```

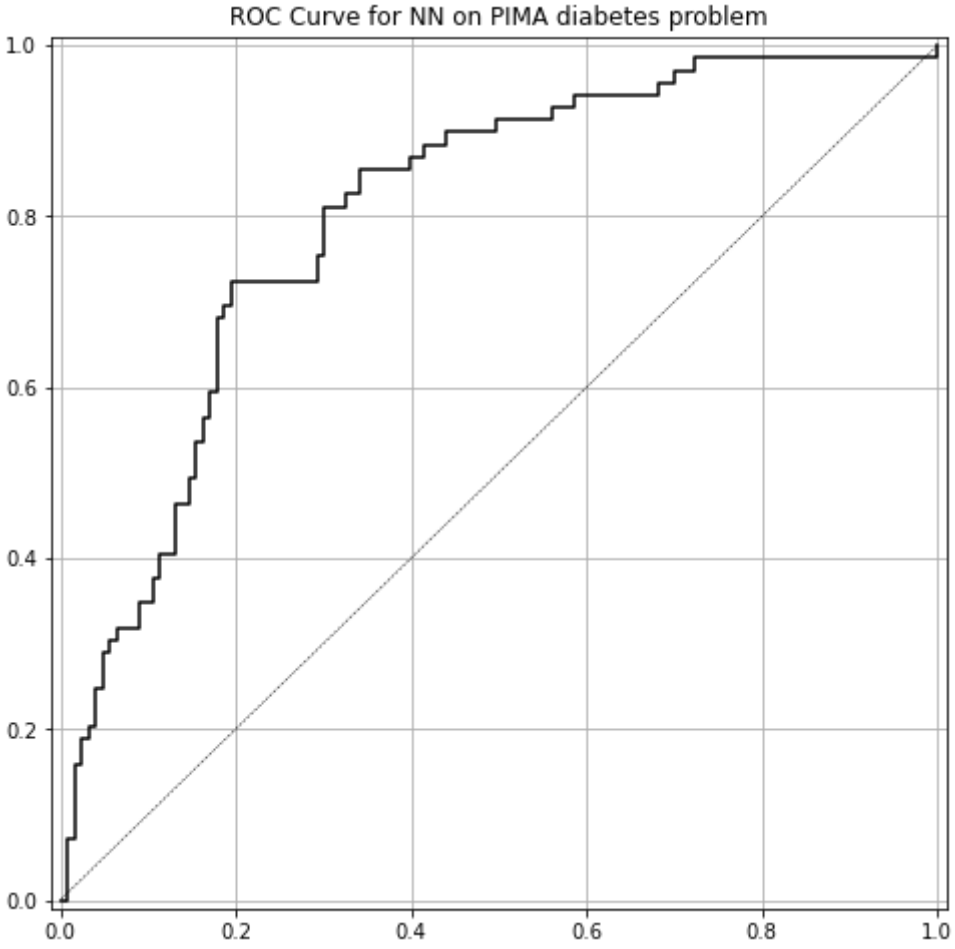
In [23]:
```
y_pred_prob[:10]
```

Out[23]:
```
array([[0.18400699],
       [0.45864904],
       [0.39152956],
       [0.3577367 ],
       [0.2979437 ],
       [0.18444127],
       [0.34327847],
       [0.3490064 ],
       [0.26480666],
       [0.18526652]], dtype=float32)
```

In [24]:
```
# Print model accuracy and roc score
print('accuracy is {:.3f}'.format(accuracy_score(y_test,y_pred_class)))
print('roc-auc is {:.3f}'.format(roc_auc_score(y_test,y_pred_prob)))
```

```
accuracy is 0.698
roc-auc is 0.803
```

In [26]:
```
def plot_roc(y_test, y_pred, model_name):
    fpr, tpr, thr = roc_curve(y_test, y_pred)
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.plot(fpr, tpr, 'k-')
    ax.plot([0, 1], [0, 1], 'k--', linewidth=.5)  # roc curve for random mo
    ax.grid(True)
    ax.set(title='ROC Curve for {} on PIMA diabetes problem'.format(model_n
           xlim=[-0.01, 1.01], ylim=[-0.01, 1.01])
plot_roc(y_test, y_pred_prob, 'NN')
```



From History object we can get Validation loss , Validation accuracy , Training Loss and Training accuracy.We can use this info to create different plots

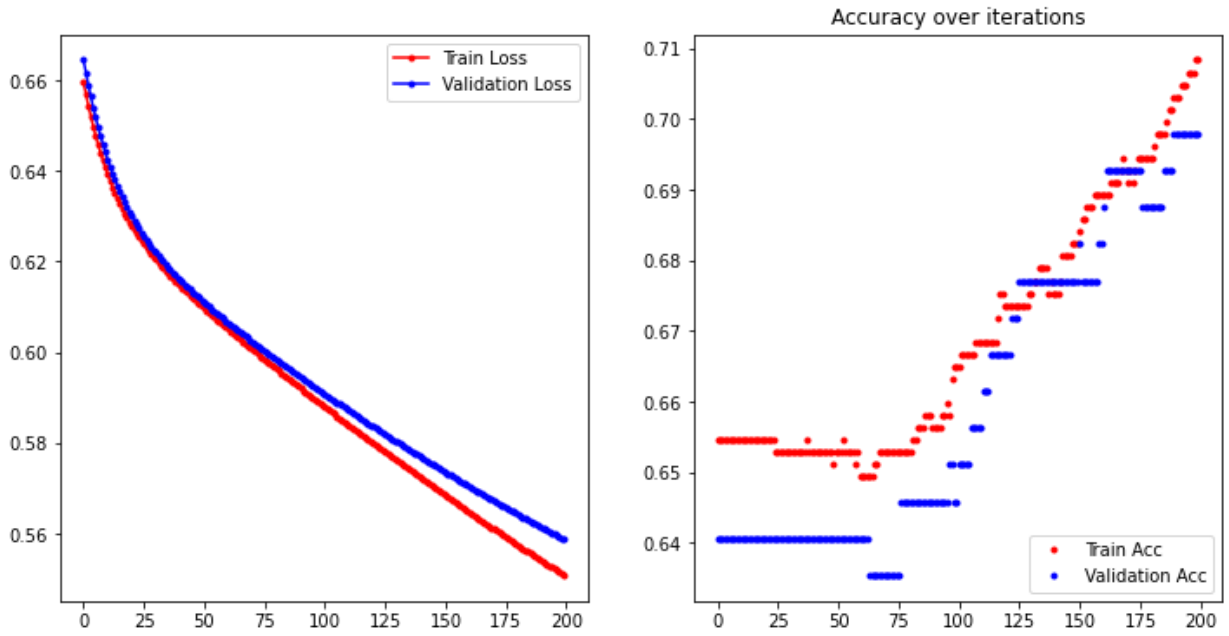
In [28]:
```
run_hist_1.history.keys()
```

Out[28]:
```
dict_keys(['val_loss', 'val_accuracy', 'loss', 'accuracy'])
```

In [36]:
```
fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot(1, 2, 1)
ax.plot(run_hist_1.history["loss"],'r', marker='.', label="Train Loss")
ax.plot(run_hist_1.history["val_loss"],'b', marker='.', label="Validation L
ax.legend()

ax = fig.add_subplot(1, 2, 2)
ax.plot(run_hist_1.history["accuracy"],'r.', label="Train Acc")
ax.plot(run_hist_1.history["val_accuracy"],'b.', label="Validation Acc")
ax.legend(loc='lower right')
ax.set_title('Accuracy over iterations')
```

Out[36]:
```
Text(0.5, 1.0, 'Accuracy over iterations')
```



Fit a new model with "ReLu" activation for final layers and Sigmoid for final activation and compare results .

In [37]:
```
### BEGIN SOLUTION
model_2 = Sequential()
model_2.add(Dense(6, input_shape=(8,), activation="relu"))
model_2.add(Dense(6,  activation="relu"))
model_2.add(Dense(1, activation="sigmoid"))

model_2.compile(SGD(lr = .003), "binary_crossentropy", metrics=["accuracy"]
run_hist_2 = model_2.fit(X_train_norm, y_train, validation_data=(X_test_nor
```

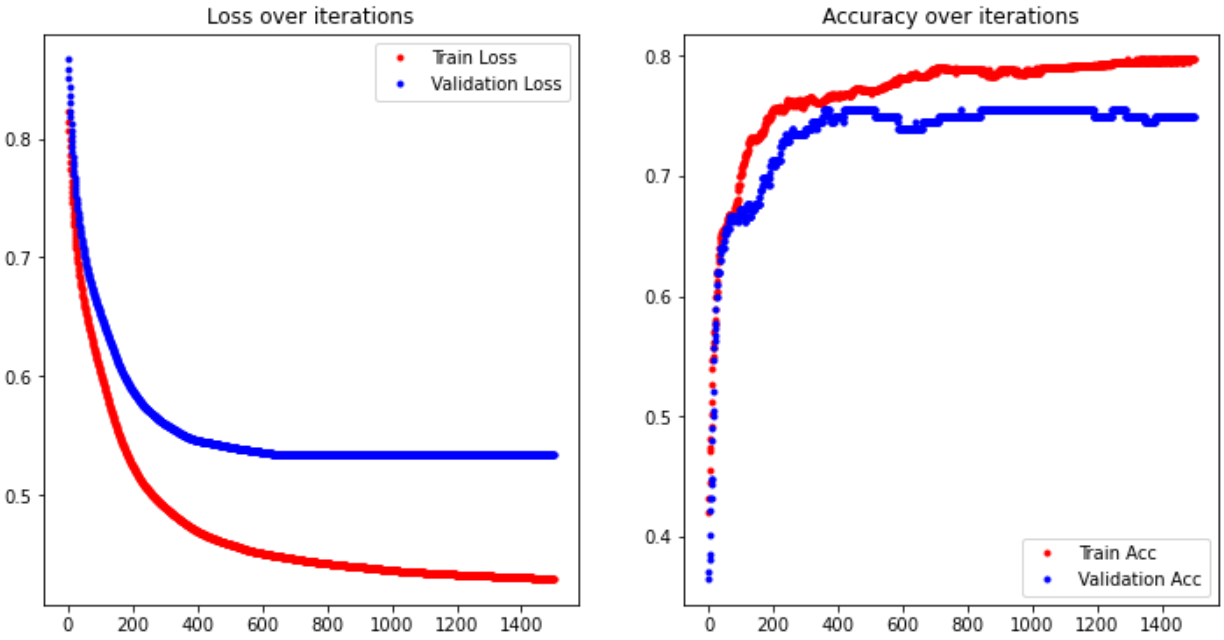
```
Train on 576 samples, validate on 192 samples
Epoch 1/1500
576/576 [==============================] - 1s 1ms/step - loss: 0.8226 - a
ccuracy: 0.4201 - val_loss: 0.8666 - val_accuracy: 0.3698
Epoch 2/1500
576/576 [==============================] - 0s 64us/step - loss: 0.8144 -
accuracy: 0.4323 - val_loss: 0.8583 - val_accuracy: 0.3646
Epoch 3/1500
576/576 [==============================] - 0s 63us/step - loss: 0.8067 -
accuracy: 0.4444 - val_loss: 0.8504 - val_accuracy: 0.3802
Epoch 4/1500
576/576 [==============================] - 0s 66us/step - loss: 0.7995 -
accuracy: 0.4549 - val_loss: 0.8432 - val_accuracy: 0.3854
Epoch 5/1500
576/576 [==============================] - 0s 60us/step - loss: 0.7928 -
accuracy: 0.4705 - val_loss: 0.8363 - val_accuracy: 0.4010
Epoch 6/1500
576/576 [==============================] - 0s 66us/step - loss: 0.7865 -
accuracy: 0.4740 - val_loss: 0.8299 - val_accuracy: 0.4219
```

In [38]:
```
n = len(run_hist_2.history["loss"])

fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot(1, 2, 1)
ax.plot(range(n), (run_hist_2.history["loss"]),'r.', label="Train Loss")
ax.plot(range(n), (run_hist_2.history["val_loss"]),'b.', label="Validation
ax.legend()
ax.set_title('Loss over iterations')

ax = fig.add_subplot(1, 2, 2)
ax.plot(range(n), (run_hist_2.history["accuracy"]),'r.', label="Train Acc")
ax.plot(range(n), (run_hist_2.history["val_accuracy"]),'b.', label="Validat
ax.legend(loc='lower right')
ax.set_title('Accuracy over iterations')
```

Out[38]:
```
Text(0.5, 1.0, 'Accuracy over iterations')
```



Summary

In NN with ReLu Validation loss increases after 800 epocs that means its overfitting . We should try different activation functions with differnent learing rate to get good accuracy score . From plots of training loss and validation loss we could make out wherther its possible to train model with more epocs to get more accuracy.

In [ ]:
```
```