Applications of Artificial Neural Network

Rini Christy

May 7, 2019

Table of Contents

1. Introduction

Artificial neural network is one of supervised machine learning techniques, consisting of multiple neurons imitating what has been observed in the human brain. The independent variables are the inputs In the Neural Network representing the incoming signals to the main neuron and acts like smell or touch in human neuron. Just like these signals they travel down the synapses, go through the main body, emerge the other side as output values. Artificial Neural Network (ANN) is used for many power demanding and highly compute intensive tasks, like for example medical diagnosis of diseases as what I'm about to do right now and for a variety of purposes from making predictions for business problems.

2. Methodology: Data Collection, Data Preprocessing, Building ANN, Evaluation & Prediction

Basically, GPU is a much better choice for neural networks, especially when I'm building deep neural networks to do deep learning. GPU has made it possible to actually train the neural network in a reasonable time, making the technique very popular in many areas. In terms of power and in terms of computation efficiency, the GPU is much more powerful because it has many more cores and it's able to run a lot more floating point calculations per second, than the CPU. During forward-propagation & back propagation of the activations of the different neurons in the neural networks using the activation functions involve parallel computations. The GPU is much more specialized for highly compute-intensive tasks and parallel computations, as in the case for neural networks. For the present study with PIMA INDIANS DIABETES dataset

obtained from Kaggle, I have several independent variables like Number of Pregnancies, Glucose level, Blood pressure, Skin thickness, Insulin, BMI, Diabetes Pedigree Function, & Age. Based on these independent variables, I need to predict which patients will be diabetic in near future. Basically this is a classification problem and I hope artificial neural networks can do an awesome job at making that kind of predictions with processing being done on my standalone laptop without the use of GPU.

The present study is divided into 3 parts: Part one will be data preprocessing, part two will only be about creating and tuning the ANN model and eventually part three involves evaluation of the model and prediction on new observations.

2.1 Part 1: Data Preprocessing

It's very important to prepare the data correctly to build a future deep learning model. The data set I obtained from Kaggle is information of patients in a diagnostic center. This dataset describes the medical records for Pima Indians and whether or not each patient will have an onset of diabetes within five years. I have some independent variables which are some information about patients in a medical diagnosis center and I am trying to predict a binary outcome for the dependent variable, which is either “one” if the patient is diagnosed with diabetes and zero if the patient is non diabetic and hence this is a classification problem with 8 independent variables as listed below.
Fields description follows:

1. Pregnancies = Number of times pregnant
2. Glucose = Plasma glucose concentration a 2 hours in an oral glucose tolerance test
3. BloodPressure = Diastolic blood pressure (mm Hg)
4. SkinThickness = Triceps skin fold thickness (mm)
5. Insulin = 2-Hour serum insulin (mu U/ml)
6. BMI = Body mass index (weight in kg/(height in m)^2)
7. DiabetesPedigreeFunction = Diabetes pedigree function
8. Age = Age (years)
9. Class = Class variable (1: tested positive for diabetes, 0: tested negative for diabetes)

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

The key thing to understand here is that all these variables here are independent variables and the last column is the dependent variable. And so, I am trying to predict the results of this dependent variable from the information of these independent variables with the artificial neural network. The next thing that I have to do is to create the matrix X of features and to do that I need to input the right indexes here because the matrix of features is nothing else than a matrix containing the independent variables. I am able to say which independent variables might have impact on the

dependent variable but what I don't know is which independent variable has the most impact on the dependent variable. And that's what artificial neural network will spot and will give the bigger weight in the neural network to those independent variables that have the most impact. The dependent variable vector Y is created from the last column. Y contains the binary outcomes, zero or one for all the 768 patients of the sample of this center. After importing the data set, I split the dataset into the training set and the test set with the test size to 0.2 to train the ANN on 614 observations and test its performance on 154 observations.

2.2 Part 2: Building and Tuning an ANN classifier model

I need to import the Keras library along with some modules of the keras library which are required to build the neural network. For this I use high-level neural network Keras library that wraps the libraries Theano and Tensoflow. The Keras library is an amazing library to build deep learning models like deep neural networks in a very few lines of code. Keras is a library based on Theano and Tensoflow, which means that it runs on Theano and Tensoflow libraries to build deep learning models very efficiently. For this study, I use Keras with TensorFlow backend. In addition to that, I need to import two more modules - the “Sequential” module that is required to initialize the neural network and the “Dense” module that is required to build the layers of ANN. After importing the required libraries to build the artificial neural network I’m going to start building Artificial Neural Network (ANN) model.

Building an ANN

An Artificial Neural Network is built and trained using Stochastic Gradient Descent in seven steps as described by SuperDataScience Team. The first step is to randomly initialize the weights of each of the nodes to small numbers close to zero using the “Dense” function which is taken from the “Dense” module imported previously. The next step is to input the first observation with each feature going to one input node. The number of nodes in the input layer is the number of independent variables in the matrix of features. There are 8 independent variables, and hence in the input layer there will be 8 input nodes. Then the third step is forward-propagation, so from left to right the neurons are activated by the activation function in such a way that the higher the values of the activation function is for the neuron, the more impact this neuron is going to have in the network. To define the first hidden layer an activation function is required and there are several activation functions like 'softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear' etc. and the most common one is the Rectifier Linear Unit (“relu”). The sigmoid function is really good for output layer since using the sigmoid function for the output layer will allow the model to get the probabilities of the different class. So that means that for each observations of the test set I'll get the probability that the patient will be diabetic or not in five years. By getting the probabilities using the sigmoid activation function for the output layer, I will be able to see which patients have the highest probabilities to get diabetes, so I will even be able to make a ranking of the patients ranked by their probability to become diabetic. So that can be very useful and then you can segment these patients according to their probability and according to that the doctors can decide what to do in terms of treatment protocols. So to summarize, I'll choose the rectifier activation function for the hidden layers and the sigmoid activation function for the output layer. In step four the algorithm compares the predicted result to the actual result and that generates an error and then in the step five this error is back-propagated in the neural network from right to left and all the weights are updated according to how much they are responsible for this generated error. There are several ways of updating these

weights and these ways are defined by the learning rate parameter which decides by how much the weights are updated. In step six, steps one to five are repeated either after each observation or either after a batch of observations and finally step seven, when the whole training set is passed through the ANN that makes an epoch and many more epochs can be repeated by choosing the number of epochs in the final line of code.

To the classifier that will be initialized by creating an object of the sequential class, different layers in the neural network are added using a method of this object which is the add method and Dense function. The arguments of Dense function includes the parameters like how the weights are updated, the activation function, the number of nodes to be in the hidden layer (output_dim) and the number of input nodes. I take the average of the number of nodes in the input layer and the number of nodes in the output layer as the value of output_dim. Experimenting with a technique called parameter tuning the performance of different models with the different parameters with the number of hidden layers and the number of nodes in the hidden layers can be tested and the optimal perimeters of the model can be chosen. The next argument is init, to randomly initialize the weight as small numbers close to zero and so I can randomly initialize them with a uniform function. The third argument is the activation argument which is the activation function for the hidden layer. I choose the rectifier activation function for the hidden layers and the sigmoid activation function for the output layer. The last argument is the input_dim parameter. This argument is the number of nodes in the input layer and that is the number of independent variables. It is compulsory to add this argument for the first stage. It's because the Artificial Neural Network initialized doesn't know which nodes this hidden layer here that is being created is expecting as inputs. So this argument needs to be specified here. I need to specify that it should expect 8 input nodes and these 8 input nodes are nothing else than the 8 independent variables. When some more other hidden layers are added further this first hidden layers will already be created and hence the input_dim argument for the next hidden layers are not required because since the first one will already be created, the next hidden layer will know what to expect. I will be adding a new hidden layer, even though not necessarily useful for this dataset, for two reasons. An Artificial Neural Network is supposed to be deep learning with many hidden layers and for the sake of practicing it I add more hidden layers in the neural networks.

For the second hidden layer, well it knows what to expect because the first hidden layer was created so no need to specify any input_dim parameter. As done before the activation function chosen is the rectifier function for all the hidden layers.
The next step is to add the final layer, that is, the output layer. Since I want to have probabilities for the outcome, I need to replace the rectifier activation function by the sigmoid activation function. First is the output_dim parameter that will be set as the number of classes, because here, I have two categories.

After the addition of all hidden and outer layers the next step is to compile the artificial neural network, that is basically applying stochastic gradient descent on the whole artificial neural network by the compile method which contains several parameters. The first argument is optimizer which is simply the algorithm to find the optimal set of weights in the neural networks as the weights are still only initialized. This algorithm is going to be nothing else than the stochastic gradient descent algorithm, and there are several types of stochastic gradient descent

algorithm, and a very efficient one called Adaptive Moment Estimation (Adam), is going to be the input of this optimizer parameter. The second parameter is loss, and this corresponds to the loss function within the stochastic gradient descent algorithm, that is within the Adam algorithm. Because going deeper into the mathematical details of stochastic gradient descent, it is based on a loss function that needs to be optimized to find the optimal weights. It is similar in working to the loss function of linear regression, where the loss function was the sum of the squared errors, which is the sum of the squared differences between the real value and the predicted value. A sigmoid activation function is actually a logistic regression model and going deeper into the mathematical details of stochastic gradient descent for the logistic regression model, the loss function is not the sum of squared errors like for linear regression, but is going to be a logarithmic function that is called the logarithmic loss. Since the dependent variable has a binary outcome, then this logarithmic loss function is called binary_cross entropy. The final argument is metrics argument, which is just a criterion to evaluate the model, and I use the accuracy criterion here. So basically what happens is that, when the weights are updated after each observation or after each batch of many observations, the algorithm uses this accuracy criterion to improve the model's performance. When the model is fitted to the training set & executed, the accuracy is going to increase little by little, until reaching a top accuracy.

The number of epochs is the number of times the ANN is trained on the whole training set, and thus when the stochastic gradient descent is in action, it is improving its accuracy at each round, that is at each epoch. The batch size is the number of observations after which, the weight is updated. To update the weight either after each observation passing through the ANN, or after a batch of observations the argument used is the batch size. This needs to be experimented on to find some optimal choice for the batch size and the number of epochs. Right now I will go with some fixed choice of batch size and number of epochs of 10 and 100 respectively, so that I can see the algorithm in action and watch the improvement of accuracy over the different epochs. Verbose set to zero is an option for producing only the final report omitting the detailed logging information. The ANN model built is then connected to the training set with the fit method that will fit the ANN to the training set, x train and y train, exactly like in the classification part for the Machine Learning algorithm. Finally, the model is complete & the codes are ready to be executed!

Measuring the accuracy on one single test set is not the most relevant way to evaluate the models performance because by retraining the model several times I get different accuracies not only on the training sets but also on the test set. And the solution for this is k-fold cross-validation. It will fix the problem by splitting the training set into 10 folds when K equals 10. With 10-folds ten different combinations of 9-folds are used to train a model and 1-fold to test it. That means that the model is trained and tested on ten combinations of training and test sets. And that will give a much better idea of the model performance because the average of the different accuracies of the ten evaluations and also compute the standard deviation to have a look at the variance. A good accuracy on a small variance will be the best one.

Keras wrapper will wrap K-fold cross validation by Scikit-learn, into the Keras model. So Keras Classifier is imported from keras.wrappers.scikit-learn and the K-fold cross validation function called Cross Val Score from Scikit-learn. The first argument is Build Function which is simply a python def function that builds the architecture of the artificial neural network & returns the

classifier. The next arguments are the batch size and the number of epochs. The classifier will be built through k-fold cross validation on 10 different training folds and by each time measuring the model performance on one test fold. The use of K-fold cross validation is to return a relevant measure of the accuracy of the artificial neural network on the test set and therefore the accuracies variable of the cross val score function will return the 10 accuracies of the 10 test folds that occur in K-fold cross validation with K equals 10. Cross val score need to input several arguments. The first argument is estimator which is the classifier built here with Keras classifier. The next parameter is the training set data to fit from which 10 train test folds are going to be created. The following parameter CV is basically the number of train test folds to be created when applying K-fold cross validation. The choice is pretty arbitrary but most of the time 10 folds are used. With 10 accuracies a low bias and low variance in the bias variance trade-off between the accuracies is expected. The last argument n_jobs, is the number of CPU to use to do the computation and minus one means all CPU's. The artificial neural network is going to be trained 10 times because it's going to be trained on 10 train folds. Using the n jobs parameter and setting it to minus one the model uses all the CPU's and computations go faster. It will run parallel computations for different trainings on the different train folds all at the same time. The relevance accuracy would be to take the mean of all these accuracies and then compute the variance check whether there is high variance or low variance.

```
# Importing the libraries
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from keras.models import Sequential
from keras.layers import Dense
# Function to create model, required for KerasClassifier
def build_classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 5, kernel_initializer = 'uniform', activation = 'relu', input_dim = 8))
    classifier.add(Dense(units = 5, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))
    classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
    return classifier
classifier = KerasClassifier(build_fn = build_classifier, batch_size = 10, epochs = 100, verbose=0 )
accuracies = cross_val_score(estimator = classifier, X = X_train, y = y_train, cv = 10, n_jobs = -1)
mean = accuracies.mean()
print("mean:", mean)
variance = accuracies.std()
print("variance:", variance)
```

```
Using TensorFlow backend.
```

```
mean: 0.7538868317807055
variance: 0.04247417708933399
```

The mean of accuracies 75%, is the relevance accuracy to evaluate the models performance and the variance is actually quite small, 0.04%, so that's less than 1% and therefore a very low variance.

Tuning of ANN Classifier Model

I will try to make a better artificial neural network to get a higher accuracy than 75%. The solution to find the best model to accomplish this goal is parameter tuning. For this some parameters are learnt from the model during the training while some other parameters stay fixed and these parameters are called the hyper parameters. These hyper parameters are the number of epochs, the batch size, the optimizer, the activation function, learn_rate, momentum or the number of neurons in the layers. Parameter tuning consists of finding the best values of these

hyper parameters and this is done with a technique called grid search that basically will test several combinations of these values and will eventually return the best selection, the best choice that leads to the best accuracy with k-fold cross validation. It's actually quite similar as implementing k-fold cross validation but instead of using the cross_val score function GridSearchCV class imported from sklearn.model_selection module will be used. Wrap the whole architecture in the classifier using the Keras classifier class but in this kerasclassifier class the batch size and number of epoch arguments are excluded because those are the arguments going to be tuned and these arguments will be put separately in the gridsearch object. A variable called parameters is created for the python dictionary with each key going to be the hyper parameter to be tuned and the values of the keys are going to be the values to be tried and tested for the artificial neural network. And then, all these values for all the keys will be combined and GridSearch will train the artificial neural network using k-fold cross validation to get a relevant accuracy with the different combinations of these values and in the end it will return best accuracy with the best selection of these values. This creates a dictionary with all the combinations of the hyper parameters that needs to be optimized. First let me try & tune several batch sizes, the number of epoch & the optimizer choosing 25 and 32 for batch size, 100 & 500 for epochs and finally 'SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam' for the optimizer. In the build classifier function replace the Adam optimizer by the optimizer argument that now plays the role of a variable. And so now, there is freedom to input different values that needs to be tested for the optimizer.

“gridsearch” object of the GridSearchCV class created will contain this parameter dictionary with all the hyper parameters that needs to be tuned, and the different values that needs to be tested. The gridserch object contains the estimator which is the classifier that was defined earlier and the second argument is param_grid, where I need to input the parameters dictionary that contains all the combinations that I want to try and tune each of these combinations. It will also contain the information related to k-fold cross validation. When the model is tuned with gridsearch, kfold cross validation is going to be used to evaluate the accuracy and so the scoring metric is going to be the accuracy, and the number of folds in k-fold cross validation, is going to be ten as done before for k-fold cross validation in the previous step. The gridsearch object is then fitted to the training set to find the best selection of the hyper parameters.

1. Tuning the Training Batch size, Epochs & Optimization Algorithm using "relu" activation function

```
best_parameters: {'batch_size': 25, 'epochs': 100, 'optimizer': 'Adam'}
best_accuracy: 0.7671009771986971
Best: 0.767101 using {'batch_size': 25, 'epochs': 100, 'optimizer': 'Adam'}
0.640065 (0.051902) with: {'batch_size': 25, 'epochs': 100, 'optimizer': 'SGD'}
0.749186 (0.027847) with: {'batch_size': 25, 'epochs': 100, 'optimizer': 'RMSprop'}
0.763844 (0.037596) with: {'batch_size': 25, 'epochs': 100, 'optimizer': 'Adagrad'}
0.754072 (0.043775) with: {'batch_size': 25, 'epochs': 100, 'optimizer': 'Adadelta'}
0.767101 (0.036450) with: {'batch_size': 25, 'epochs': 100, 'optimizer': 'Adam'}
0.757329 (0.035958) with: {'batch_size': 25, 'epochs': 100, 'optimizer': 'Adamax'}
0.762215 (0.022844) with: {'batch_size': 25, 'epochs': 100, 'optimizer': 'Nadam'}
0.698697 (0.056662) with: {'batch_size': 25, 'epochs': 500, 'optimizer': 'SGD'}
0.755700 (0.053267) with: {'batch_size': 25, 'epochs': 500, 'optimizer': 'RMSprop'}
0.757329 (0.031977) with: {'batch_size': 25, 'epochs': 500, 'optimizer': 'Adagrad'}
0.755700 (0.041565) with: {'batch_size': 25, 'epochs': 500, 'optimizer': 'Adadelta'}
0.752443 (0.033560) with: {'batch_size': 25, 'epochs': 500, 'optimizer': 'Adam'}
0.765472 (0.035542) with: {'batch_size': 25, 'epochs': 500, 'optimizer': 'Adamax'}
0.728013 (0.051079) with: {'batch_size': 25, 'epochs': 500, 'optimizer': 'Nadam'}
0.640065 (0.051902) with: {'batch_size': 32, 'epochs': 100, 'optimizer': 'SGD'}
0.762215 (0.038026) with: {'batch_size': 32, 'epochs': 100, 'optimizer': 'RMSprop'}
0.758958 (0.039927) with: {'batch_size': 32, 'epochs': 100, 'optimizer': 'Adagrad'}
0.755700 (0.038122) with: {'batch_size': 32, 'epochs': 100, 'optimizer': 'Adadelta'}
0.758958 (0.034829) with: {'batch_size': 32, 'epochs': 100, 'optimizer': 'Adam'}
0.757329 (0.028122) with: {'batch_size': 32, 'epochs': 100, 'optimizer': 'Adamax'}
0.757329 (0.041532) with: {'batch_size': 32, 'epochs': 100, 'optimizer': 'Nadam'}
0.664495 (0.067769) with: {'batch_size': 32, 'epochs': 500, 'optimizer': 'SGD'}
0.760586 (0.039559) with: {'batch_size': 32, 'epochs': 500, 'optimizer': 'RMSprop'}
0.755700 (0.035695) with: {'batch_size': 32, 'epochs': 500, 'optimizer': 'Adagrad'}
0.755700 (0.030434) with: {'batch_size': 32, 'epochs': 500, 'optimizer': 'Adadelta'}
0.742671 (0.043867) with: {'batch_size': 32, 'epochs': 500, 'optimizer': 'Adam'}
0.754072 (0.030546) with: {'batch_size': 32, 'epochs': 500, 'optimizer': 'Adamax'}
0.755700 (0.037045) with: {'batch_size': 32, 'epochs': 500, 'optimizer': 'Nadam'}
```

The best results were generated with a batch size of 25, epochs 100 with the optimizer "Adam" achieving a performance of about 76.7%.

2. Tuning the Neuron Activation Function using "Adam" optimizer, Batch size of 25 & Epochs 100

The activation function controls the non-linearity of individual neurons. The optimum function from different activation functions available in Keras can be found by using these functions in the hidden layer, as a sigmoid activation function in the output layer for the binary classification problem is required.

```
best_parameters: {'activation': 'tanh'}
best_accuracy: 0.7671009771986971
Best: 0.767101 using {'activation': 'tanh'}
0.763844 (0.027804) with: {'activation': 'softmax'}
0.750814 (0.028070) with: {'activation': 'softplus'}
0.760586 (0.040186) with: {'activation': 'softsign'}
0.752443 (0.033412) with: {'activation': 'relu'}
0.767101 (0.039260) with: {'activation': 'tanh'}
0.750814 (0.042023) with: {'activation': 'sigmoid'}
0.732899 (0.081227) with: {'activation': 'hard_sigmoid'}
0.754072 (0.050285) with: {'activation': 'linear'}
```

The best fit obtained is with 'tanh' activation achieving an accuracy of 76.7%

3. Tuning the Neuron Activation Function using "RMSprop" optimizer

```
best_parameters: {'activation': 'relu'}
best_accuracy: 0.7736156351791531
Best: 0.773616 using {'activation': 'relu'}
0.762215 (0.034797) with: {'activation': 'softmax'}
0.745928 (0.043180) with: {'activation': 'softplus'}
0.758958 (0.039209) with: {'activation': 'softsign'}
0.773616 (0.034452) with: {'activation': 'relu'}
0.762215 (0.039688) with: {'activation': 'tanh'}
0.640065 (0.051902) with: {'activation': 'sigmoid'}
0.640065 (0.051902) with: {'activation': 'hard_sigmoid'}
0.762215 (0.048573) with: {'activation': 'linear'}
```

Using "RMSprop" optimizer the best accuracy of 77% achieved is with "relu" activation

2.3 Part 3: Evaluation & Prediction

The best parameters selected by grid search are batch size of 25, epoch of 100, "relu" activation and an "RMSprop" optimizer and so it's with these parameters that the model managed to get the 77% accuracy. Thus artificial neural network is trained on the training set and now time to make the predictions on the test set. Making predictions on the test set will be interesting to evaluate the performance of the model on new observations and to see if the accuracy of prediction is going to be as good as the one obtained on the training set.

```
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from keras.models import Sequential
from keras.layers import Dense
def build_classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 5, kernel_initializer = 'uniform', activation = 'relu', input_dim = 8))
    classifier.add(Dense(units = 5, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))
    classifier.compile(optimizer = 'RMSprop', loss = 'binary_crossentropy', metrics = ['accuracy'])
    return classifier
classifier = KerasClassifier(build_fn = build_classifier, epochs=100, batch_size=25, verbose=0)
accuracies = cross_val_score(estimator = classifier, X = X_train, y = y_train, cv = 10, n_jobs = -1)
mean = accuracies.mean()
print("mean:", mean)
variance = accuracies.std()
print("variance:", variance)
```

```
mean: 0.7621364380098536
variance: 0.039127227763600975
```

Confusion Matrix

The accuracy on new observations can be visualized using confusion matrix. The results of confusion matrix give the number of correct predictions and the number of incorrect predictions, and then eventually, the accuracy on new observations.

```
# Predicting the Test set results
classifier.fit(X_train, y_train, batch_size = 25, epochs = 100)
y_pred = classifier.predict(X_test)
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

```
array([[98,  9],
       [19, 28]], dtype=int64)
```

Out of 154 new observations which were not used to train the artificial neural network, the model rendered 126 correct predictions, and 28 incorrect predictions, which looks really good. The accuracy is the number of correct predictions divided by the total number of predictions; thus the model achieved an accuracy of 82%.
The model can be tuned further to get an even better accuracy.

4. Tuning of kernel_initializer using "RMSprop" optimizer & "relu" activation for hidden layer & "sigmoid" for outer layer

```
best_parameters: {'init_mode': 'he_normal'}
best_accuracy: 0.7687296416938111
Best: 0.768730 using {'init_mode': 'he_normal'}
0.757329 (0.035512) with: {'init_mode': 'uniform'}
0.744300 (0.036388) with: {'init_mode': 'lecun_uniform'}
0.760586 (0.034111) with: {'init_mode': 'normal'}
0.640065 (0.051902) with: {'init_mode': 'zero'}
0.750814 (0.047247) with: {'init_mode': 'glorot_normal'}
0.755700 (0.030939) with: {'init_mode': 'glorot_uniform'}
0.768730 (0.035791) with: {'init_mode': 'he_normal'}
0.760586 (0.050308) with: {'init_mode': 'he_uniform'}
```

The best results were achieved with a "he_normal" kernel initializer achieving a performance of about 77%.

5. Tuning of learn rate using "Adam" optimizer & "relu" activation for hidden layer & "sigmoid" for outer layer

Learning rate is a hyper-parameter that controls how much the weights are adjusted in the neural network with respect to the loss gradient. Lower the value, smaller the weight adjustments and hence longer the time taken to reach the global minimum error.

```
best_parameters: {'learn_rate': 0.05}
best_accuracy: 0.7638436482084691
Best: 0.763844 using {'learn_rate': 0.05}
0.755700 (0.039087) with: {'learn_rate': 0.025}
0.763844 (0.050420) with: {'learn_rate': 0.05}
0.752443 (0.040021) with: {'learn_rate': 0.1}
0.744300 (0.028700) with: {'learn_rate': 0.2}
0.745928 (0.057736) with: {'learn_rate': 0.3}
```

The optimum learning rate obtained is 0.05 with an accuracy of 76%.

Prediction on the test sets can be repeated applying the tuned optimum parameters like "he_normal" kernel_initializer, "relu" activation function for the hidden layer, "sigmoid" for the outer layer and RMSprop optimizer.

```
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from keras.models import Sequential
from keras.layers import Dense
def build_classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 5, kernel_initializer = 'he_normal', activation = 'relu', input_dim = 8))
    classifier.add(Dense(units = 5, kernel_initializer = 'he_normal', activation = 'relu'))
    classifier.add(Dense(units = 1, kernel_initializer = 'he_normal', activation = 'sigmoid'))
    classifier.compile(optimizer = 'RMSprop', loss = 'binary_crossentropy', metrics = ['accuracy'])
    return classifier
classifier = KerasClassifier(build_fn = build_classifier, epochs=100, batch_size=25, verbose=0)
accuracies = cross_val_score(estimator = classifier, X = X_train, y = y_train, cv = 10, n_jobs = -1)
mean = accuracies.mean()
print("mean:", mean)
variance = accuracies.std()
print("variance:", variance)
# Predicting the Test set results
classifier.fit(X_train, y_train, batch_size = 25, epochs = 100)
y_pred = classifier.predict(X_test)
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

```
mean: 0.7458223186728816
variance: 0.03481654096826264
```

```
array([[98,  9],
       [17, 30]], dtype=int64)
```

The model has achieved 83% accuracy on the test set although with some compromise on the accuracy of the train set.

3. Conclusion

By training the network several hundred times, it has grown to predict all the unknown parameters with uncertainties of about a few percent. A good accuracy of 83% on the test set validates the model. The diagnostic center could use this validated model on all the patients of this center, and by ranking the probabilities from the highest to the lowest, it gets a ranking of the patients most likely to become diabetic in near future. So then, for example, the doctor can have a look at the 10% highest probabilities of their patients to the likelihood of diabetes and or make segments of different classes, and then analyze in more depth thanks to data-mining techniques, to understand why the patients of this segment are the most likely to get diabetes. Thus the artificial neural network that was just made creates a lot of added value for the center, because by targeting these patients, the treatment can be adjusted and with some lifestyle modification the likelihood of onset of diabetes can be prevented well in advance.

4. References

1. https://www.kaggle.com/kumargh/pimaindiansdiabetescsv
2. https://www.superdatascience.com/blogs/the-ultimate-guide-to-artificial-neural-networks-ann
3. https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/
4. Neural network-based preprocessing to estimate the parameters of the X-ray emission of a single-temperature thermal plasma https://arxiv.org/pdf/1801.06015.pdf