Improve Model Accuracy With Grid Search

In this tutorial, you will discover how to tune Keras hyperparameters using grid searching strategy. A hyperparameter is a parameter that is set before the learning process begins. These parameters are tunable and can directly affect how well a model trains. Some examples of hyperparameters in deep learning are:

- the learning rate
- the number of Epochs
- the parameters of the optimizer (RMSprop, Adam…)
- the regularization parameter
- the number of neurons in the layer of a neural network
- the number of layers of a neural network

In other words, the hyperparameter of a neural network are pretty much all the parameters but the weights and the biases.

1. Grid Search Deep Learning Model Parameters

Often the general effects of hyperparameters on a model are known, but how to find the best set of hyperparameters for a given dataset may be challenging. There are general heuristics or rules of thumb for configuring hyperparameters.

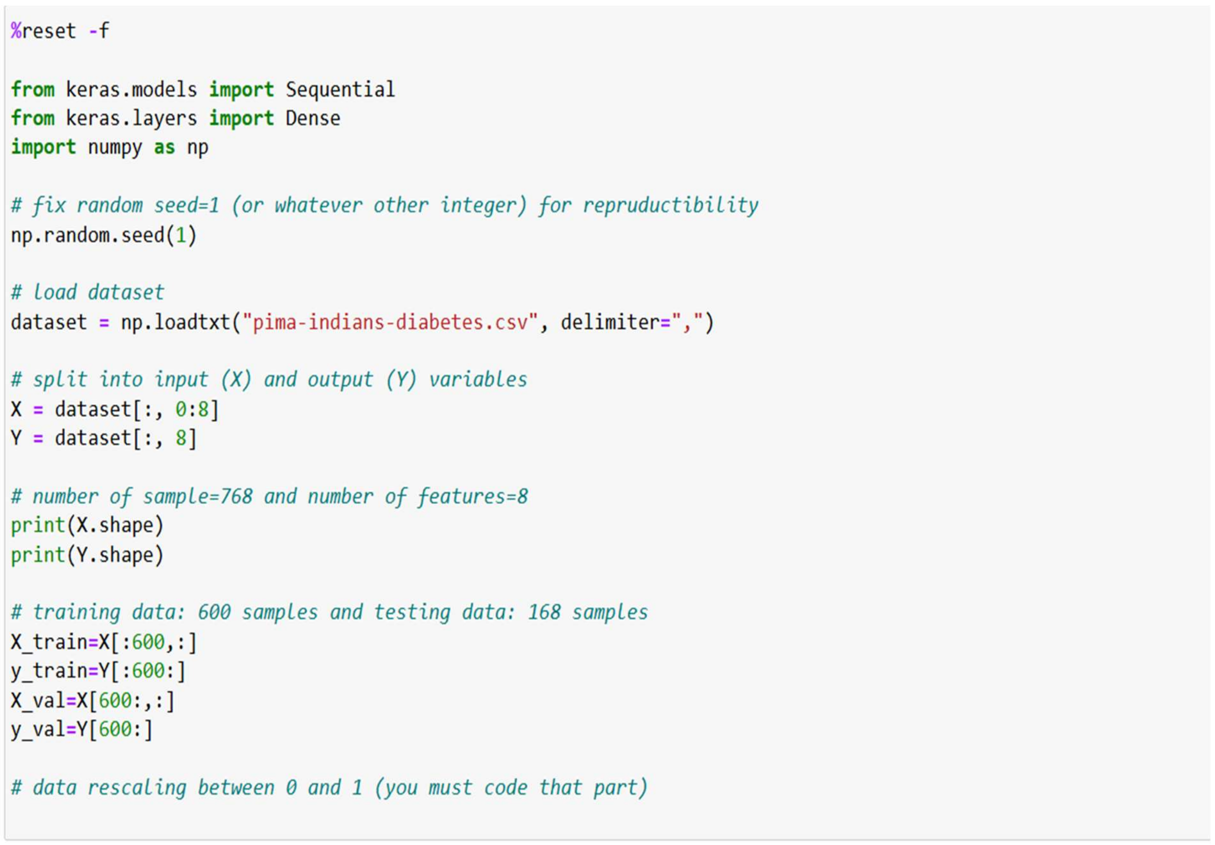
A better approach is to objectively search different values for model hyperparameters and choose a subset that results in a model that achieves the best performance on a given dataset. This is called hyperparameter optimization or hyperparameter tuning. The result of a hyperparameter optimization is a single set of well-performing hyperparameters that you can use to configure your model. Here are some common strategies for optimizing hyperparameters:

Grid Search: Search a set of manually predefined hyperparameters for the best performing hyperparameter (this is the traditional method).

Random Search: Similar to grid search but replaces the exhaustive search with random search.

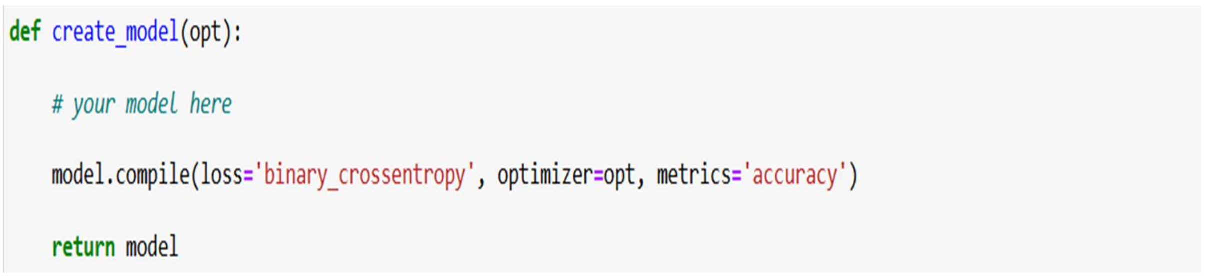
Bayesian Optimization: Builds a probabilistic model of the function mapping from hyperparameter values to the target evaluated on a validation set.

In this tutorial we will use the same data set as before (pima Indians diabetes data set) and implement grid search to find hyperparameters values. After importing the usual Python libraries, we start by downloading the data and split the data into training data set (sample of size 600) and validating data set (sample of size 168). Of course, one should not forget to **rescale the input data**. If you do not rescale the input data, your neural network will perform very poorly!



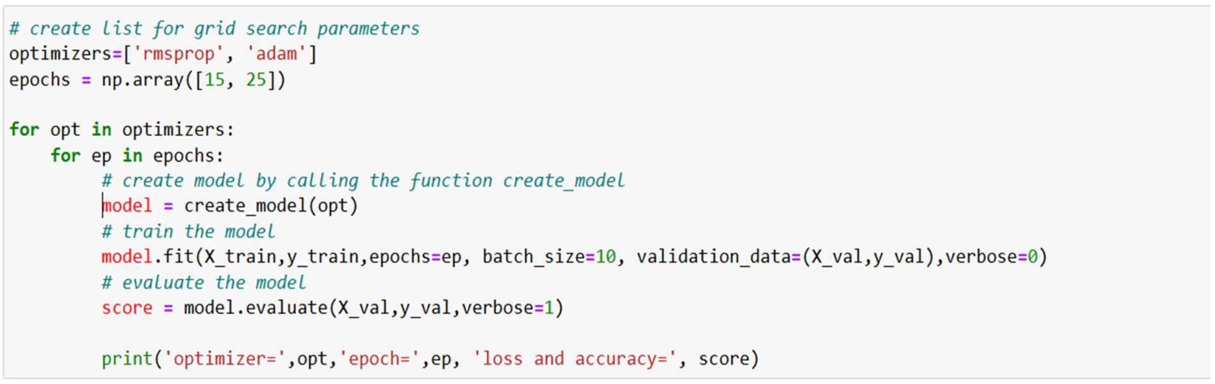
```
%reset -f

from keras.models import Sequential
from keras.layers import Dense
import numpy as np

# fix random seed=1 (or whatever other integer) for repruductibility
np.random.seed(1)

# load dataset
dataset = np.loadtxt("pima-indians-diabetes.csv", delimiter=",")

# split into input (X) and output (Y) variables
X = dataset[:, 0:8]
Y = dataset[:, 8]

# number of sample=768 and number of features=8
print(X.shape)
print(Y.shape)

# training data: 600 samples and testing data: 168 samples
X_train=X[:600,:]
y_train=Y[:600:]
X_val=X[600:,:]
y_val=Y[600:]

# data rescaling between 0 and 1 (you must code that part)
```

Then, we create a function that builds a basic neural network. Note that the function is slightly different from the one of the previous tutorial since we pass the argument *opt* to it. This argument is a string that specifies the optimizer to be used (for example, *opt* could be *‘rmsprop’* or *‘adam’*).



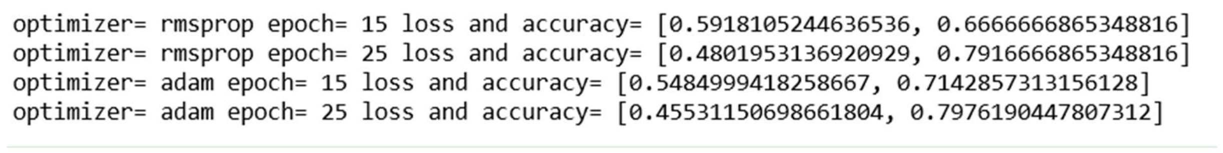
```
def create_model(opt):

    # your model here

    model.compile(loss='binary_crossentropy', optimizer=opt, metrics='accuracy')

    return model
```

In the main program, we will perform a grid search with two optimizers *['rmsprop', 'adam']* and two number of epochs *[15, 25]* that are stored into a list. The grid search will then build and train models for all the combinations of parameters of the two lists, i.e there will be four models built and tested.
Then we compute the score (evaluation of the loss and the accuracy of the model after the last epoch) and print it to screen. The *verbose=0* argument prevents from having too much information printed on the screen. Here we do not let Keras choose the training/validation data by just giving the proportion of the validation data, but we impose the validation data by setting *validation_data=(X_val,y_val)*.



```python
# create list for grid search parameters
optimizers=['rmsprop', 'adam']
epochs = np.array([15, 25])

for opt in optimizers:
    for ep in epochs:
        # create model by calling the function create_model
        model = create_model(opt)
        # train the model
        model.fit(X_train,y_train,epochs=ep, batch_size=10, validation_data=(X_val,y_val),verbose=0)
        # evaluate the model
        score = model.evaluate(X_val,y_val,verbose=1)

        print('optimizer=',opt,'epoch=',ep, 'loss and accuracy=', score)
```

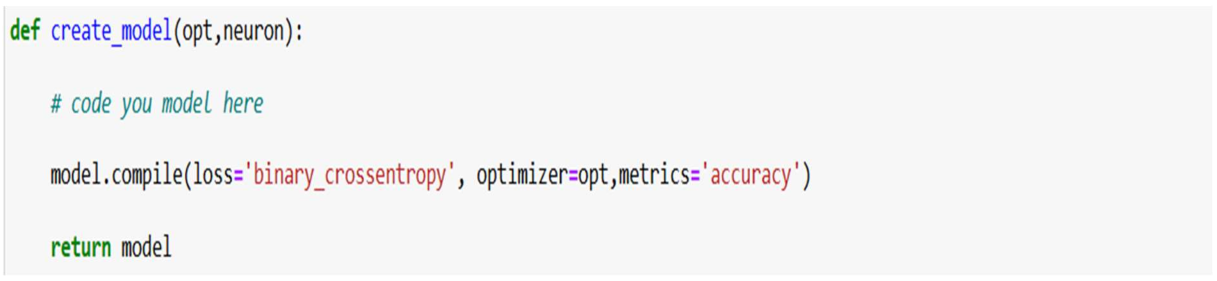
Typical results are shown below. The worst model is built with 15 epochs using RMSprop optimizer and tops at 66.6% of good classification. This is just slightly better than the baseline result (65%). The best model is built with 25 epochs using Adam optimizer with almost 80% of good classification. Results are shown below.



```
optimizer= rmsprop epoch= 15 loss and accuracy= [0.5918105244636536, 0.6666666865348816]
optimizer= rmsprop epoch= 25 loss and accuracy= [0.4801953136920929, 0.7916666865348816]
optimizer= adam epoch= 15 loss and accuracy= [0.5484999418258667, 0.7142857313156128]
optimizer= adam epoch= 25 loss and accuracy= [0.45531150698661804, 0.7976190447807312]
```

2. Grid Search With Other Parameters

The grid search can be performed with more than two parameters. All we need to change is adding arguments to the function *create_model*; define the lists of parameters in the main program; and add *for loops* in the main program, if needed.
Assume that, on top of the two previous parameters, we plan to make a grid search on the number of neurons of two layers of our neural network. For our model, the last layer of the network must be of dimension 1 since we are doing a binary classification (i.e. the output must be a scalar between 0 and 1). Therefore, we cannot modify the number of neurons of the last layer. For the two other layers, we can specify any number of neurons (in the previous example, it was set to 12 neurons for the first layer and 8 for the second layer).
Here is how the function *create_model* should be modified (assuming neuron is a numpy vector of size 2):



```python
def create_model(opt,neuron):

    # code you model here

    model.compile(loss='binary_crossentropy', optimizer=opt,metrics='accuracy')

    return model
```

And the main program now has three nested loops. The number of models that will be created in this example of grid search is 2x2x2=8. The number of neurons in the first two layers of the models will be *[12, 8]* or *[24,16]* (assuming your model has two hidden layers).

```python
# create list for grid search parameters
optimizers=['rmsprop', 'adam']
epochs = np.array([15, 25])
neuron = np.array([[12,8],[24,16]])

for neur in neuron:
    for opt in optimizers:
        for ep in epochs:
            # create model by calling the function create_model
            model = create_model(opt,neur)
            # train the model
            model.fit(X_train,y_train,epochs=ep, batch_size=10, validation_data=(X_val,y_val),verbose=0)
            # evaluate the model
            score = model.evaluate(X_val,y_val,verbose=0)

            print('neuron=',neur,'opt=',opt,'epoch=',ep,'accuracy=', score[1])
```

It is also possible to set the number of hidden layers as a parameter. In that case, the function *create_model* should be modified as follows:

```python
def create_model(opt,neuron):

    n=len(neuron)

    model=Sequential()
    model.add(Dense(neuron[0], input_dim=8, activation='relu'))

    for i in range(1,n):
        model.add(Dense(neuron[i], activation='relu'))

    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=opt,metrics='accuracy')

    return model
```

In that case, the number of hidden layers is given by the length of the vector *neuron*. Typical results are shown below:

```
neuron= [12  8] opt= rmsprop epoch= 15 accuracy= 0.6607142686843872
neuron= [12  8] opt= rmsprop epoch= 25 accuracy= 0.8035714030265808
neuron= [12  8] opt= adam epoch= 15 accuracy= 0.7559523582458496
neuron= [12  8] opt= adam epoch= 25 accuracy= 0.7976190447807312
neuron= [24 16] opt= rmsprop epoch= 15 accuracy= 0.773809552192688
neuron= [24 16] opt= rmsprop epoch= 25 accuracy= 0.773809552192688
neuron= [24 16] opt= adam epoch= 15 accuracy= 0.761904776096344
neuron= [24 16] opt= adam epoch= 25 accuracy= 0.7916666865348816
```

3. Your work.

First, you should implement and test the codes of sections 1 and 2. Then, you must experiment by changing the list of parameters (length and values). Amongst the parameters we have not tested yet, are the activations functions. You should go to the Keras web page and find the possible activation functions you can use for this problem. Then, put them in a list to perform a grid search together with other parameters. Can you do better than 80% of good classification with some set of parameters?

Exercise:

In general, it is not a good idea to set the number of epochs as a parameter in the grid search. Instead, this hyperparameter should be set using the *callbacks* Keras object that we saw in the last tutorial. Make a grid search in conjunction with Keras callbacks.