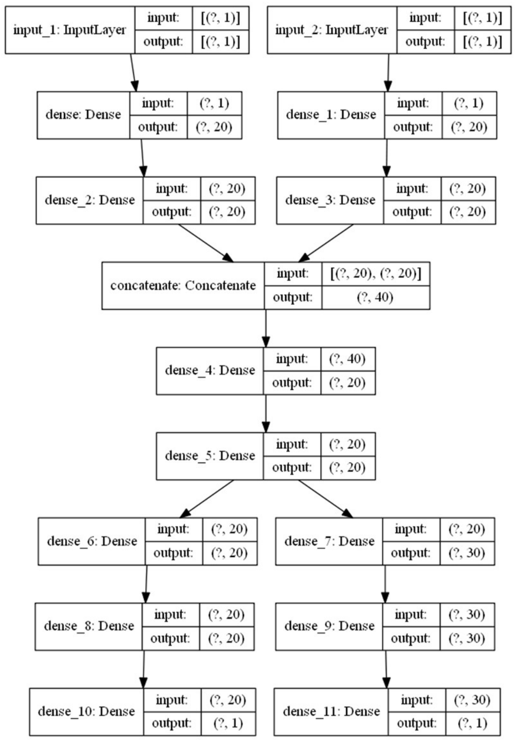
Using Keras Functional API to design complex neural networks (part II)

1. Multiple inputs and multiple outputs network

In the previous tutorial, we saw how to build a neural network with several inputs and another one with several outputs. In this section we'll see merge the two ideas to build a neural network with both multiple inputs and multiple outputs. The architecture of the neural network we'll build is as follows:



input_1: InputLayer | input: [(?, 1)] | output: [(?, 1)]
input_2: InputLayer | input: [(?, 1)] | output: [(?, 1)]
dense: Dense | input: (?, 1) | output: (?, 20)
dense_1: Dense | input: (?, 1) | output: (?, 20)
dense_2: Dense | input: (?, 20) | output: (?, 20)
dense_3: Dense | input: (?, 20) | output: (?, 20)
concatenate: Concatenate | input: [(?, 20), (?, 20)] | output: (?, 40)
dense_4: Dense | input: (?, 40) | output: (?, 20)
dense_5: Dense | input: (?, 20) | output: (?, 20)
dense_6: Dense | input: (?, 20) | output: (?, 20)
dense_7: Dense | input: (?, 20) | output: (?, 30)
dense_8: Dense | input: (?, 20) | output: (?, 20)
dense_9: Dense | input: (?, 30) | output: (?, 30)
dense_10: Dense | input: (?, 20) | output: (?, 1)
dense_11: Dense | input: (?, 30) | output: (?, 1)

In one output, we want our neural network to learn a sinus function and on the other output, we want to the network to learn an exponential function. In the last tutorial, we saw that feeding x_i as input to a single input network, it was difficult to get a sinus in one output and an exponential in the other output. In this new architecture, we'll try to feed x_i in one input and x_i^2 to the other input and see if it can help learning two different functions as the outputs.
First, we import the usual libraries and generate the training and validation data: *X_train1* and *X_train2* for the left and right input of the network, respectively (same for *X_val1* and *X_val2*). Remember that each of those matrices must have the size *(Number of samples, Number of features)*. The number of samples must be the same for all inputs and all outputs since for one input sample, there should be only one output sample. On the other hand, the number of features can be different at inputs and the dimension of the target vector can also be different from one output to the other. The values of the sinus function are stored in the matrix *y_train1* (resp. *y_val1*) and the values of the exponential function are stored in the matrix *y_train2* (resp. *y_val2)*.

Here is the first part of the code:

```
# import librairies
import numpy as np
from keras.models import Model
from keras.layers import Dense
from keras import Input
from keras.layers.merge import concatenate
import matplotlib.pyplot as plt

# fix random seed for reproductability and generate data
seed=7
np.random.seed(seed)

# create training and validation data
N=1000
X_train1=np.linspace(0.0, 1.0, N)
X_train2=X_train1*X_train1
y_train1=np.sin(2*np.pi*X_train1)
y_train2=np.exp(X_train1)

X_val1=np.random.rand(100)
X_val1=np.sort(X_val1)
X_val2=X_val1*X_val1
y_val1=np.sin(2*np.pi*X_val1)
y_val2=np.exp(X_val1)
```

Then, building the model follows the same lines as what we saw before when creating multi inputs or multi outputs model. It is not necessary to have the same number of layers or neurons in each branch of the network. At some points, we need to merge branches of the network. Keras offers several options: concatenate, add, subtract, multiply…and so on. Here we've decided to concatenate *x2_left* and *x2_right* resulting in a tensor *x3* whose shape is the sum of the shape of *x2_left* and the shape of *x2_right*. And lastly, the model is defined by its two inputs tensors and two output tensors.

```
# input layers
x0_left = Input(shape=(1,))
x0_right = Input(shape=(1,))

# build your model below


# last layers
x8_left = Dense(1)(x7_left)
x8_right = Dense(1)(x7_right)

model =  Model(inputs=[x0_left,x0_right], outputs=[x8_left,x8_right])
```

For the model fitting, we need to specify the training and testing data used to train the model. Since our model has two inputs and two outputs, the data are given under the form of lists of two tensors for the input and the output.

```
# compile the model and compute the parameters of the model
model.compile(optimizer='rmsprop',loss=['mse','mse'],loss_weights=[1,1],metrics='mse')
model.fit([X_train1,X_train2],[y_train1,y_train2],epochs=300, batch_size=32, validation_data=([X_val1,X_val2],[y_val1,y_val2]))
```

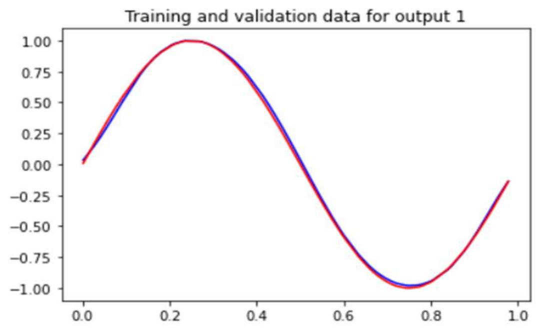
Now it's time to test our model on data it has never seen. Since there are two inputs, me must feed the *predict()* method with a list of two tensors. Similarly, since there are two outputs, we affect the result in a list of two tensors.

```
[yp_sin, yp_exp]=model.predict([X_val1,X_val2])
```

And let's compare the exact sinus with the one given by the neural network:

```
plt.plot(X_val1,yp_sin, 'b')
plt.plot(X_val1,y_val1, 'r')
plt.title('Training and validation data for output 1')
```

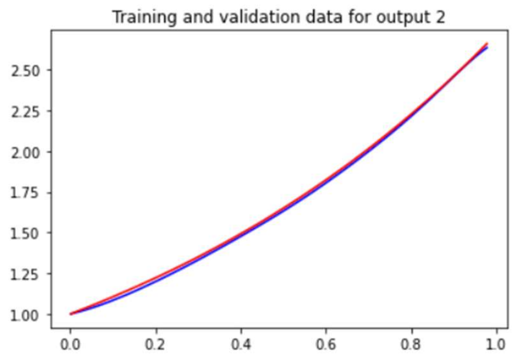
```
Text(0.5, 1.0, 'Training and validation data for output 1')
```



Seems like the second input we have added has brought some improvement on the sinus. Let's see now how the exponential is:

```
plt.plot(X_val1,yp_exp, 'b')
plt.plot(X_val1,y_val2, 'r')
plt.title('Training and validation data for output 2')
```

```
Text(0.5, 1.0, 'Training and validation data for output 2')
```



Not bad either!

2. Residual networks (ResNet)

If we look back in the history of neural networks, twenty years ago, networks that had more than about ten layers were suffering from the vanishing gradient problem (the gradient used during the training was fading as we were getting close to the first layers of the network). Therefore, it was not possible to train very deep neural network. The use of new activation functions (such as ReLu) pushed the limit up to a few dozens of layers. Nowadays, it is possible to train neural networks with hundreds of layers. How is that possible? The answer is: thanks to residual networks (ResNet)! ResNet (first introduced in 2015), are the ultimate development of neural networks and all the modern deep learning architectures involve ResNet.
ResNet utilize skip connections, or shortcuts to jump over some layers. Typical ResNet models are implemented with double- or triple- layer jumps. The second image of the previous tutorial is a typical example of ResNet. With Keras, they can be generated very easily as we will see in this section.
Let's try to get a neural network with 12 layers (1 input layer + 10 hidden layers + 1 output layer) learn a sinus function. We import the usual libraries and we generate the dataset.

```
%reset -f

import numpy as np
from keras import models
from keras import Input
from keras.models import Model
from keras.layers import Dense
from keras import layers
from keras import optimizers
import matplotlib.pyplot as plt
```

```
# fix random seed for reproductability
seed=7
np.random.seed(seed)

# generate training data
N=1000
X_train=np.linspace(0.0, 1.0, N)
y_train=np.sin(12*np.pi*X_train)

# generate validation data
X_val=np.random.rand(100)
X_val=np.sort(X_val)
y_val=np.sin(12*np.pi*X_val)
```

The model can be created with a do loop to avoid repeating ten times the following line

```
    x = Dense(10, activation='relu')(x)
```

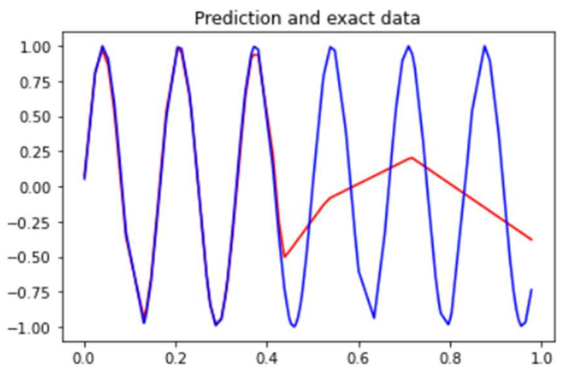
The designing of the neural network is done below.

```
x0 = Input(shape=(1,))

x = Dense(10, input_dim=1, activation='relu')(x0)

for i in range(10):
    x = Dense(10, activation='relu')(x)

x1 = Dense(1)(x)

model =  Model(inputs=x0, outputs=x1)

model.summary()
```

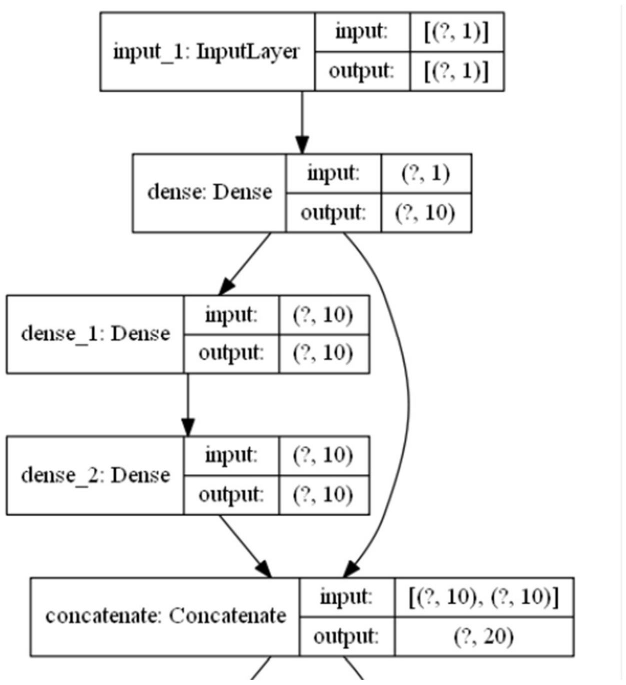
Then, we can compile and fit the model and compare the exact solution with the approximate solution.

```
# compile and fit the model
model.compile(optimizer='adam',loss='mse',metrics=['mse'])
history=model.fit(X_train,y_train,epochs=200, batch_size=32, verbose=0, validation_data=(X_val,y_val))

# compute model prediction on the validation data
y_p=model.predict(X_val)

# plot the prediction (in red) and the exact values (in blue)
plt.plot(X_val,y_p,'r-')
plt.plot(X_val,y_val,'b-')
plt.title('Prediction and exact data')
```

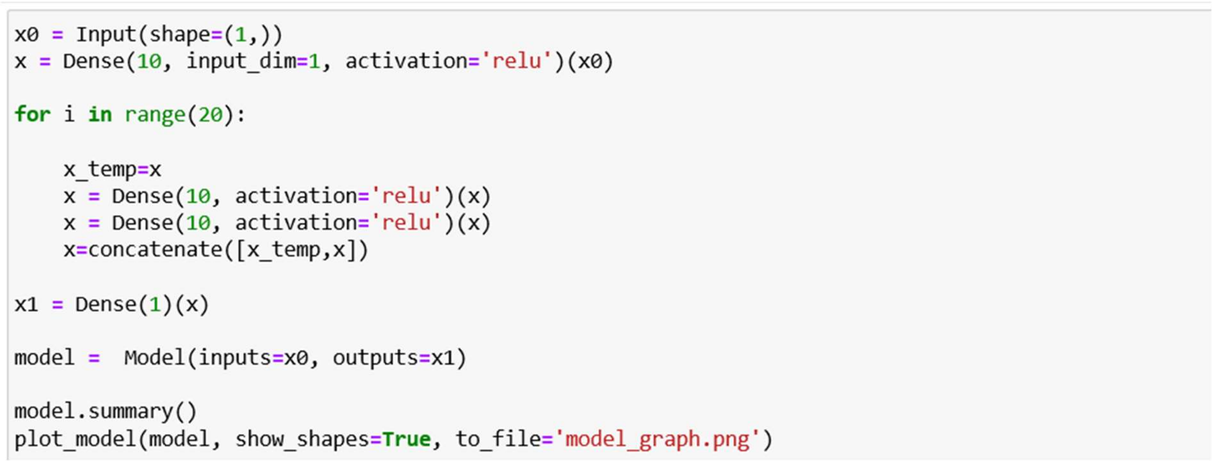
```
Text(0.5, 1.0, 'Prediction and exact data')
```



As we can see, with 12 layers, the neural network fails to properly learn the sinus function. Remember that in a previous tutorial, you were able to model this function with a shallow neural network. Adding more layers has worsen the results.
Let’s treat the same problem with a deep neural network having 42 layers and residual network connections. On top of the input and output layers, the network will be composed of twenty identical patterns. Each pattern is composer of two fully connected layers and a residual connection between the input and the output of the pattern. At the output of the pattern, we need to concatenate the two tensors coming from the residual network and the fully connected layers. We could have merged the two tensors in a different way: add, subtract, multiply… The following picture shows the first layer and the first pattern.



Coding this neural network is extremely simple: the do loop creates twenty identical patterns. A temporary tensor *x_temp* is required to do the residual connection that will skip two layers. Here is the code, together with the *plot_model* instruction for plotting your ~~piece of art~~ neural network.

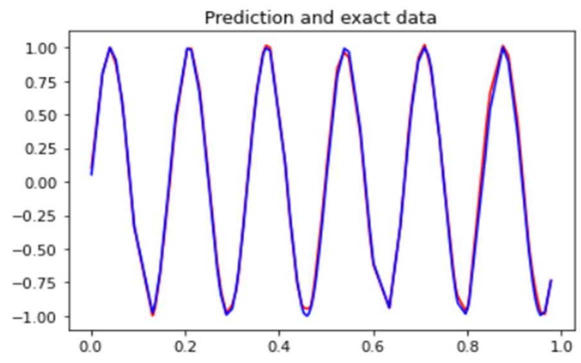


```
x0 = Input(shape=(1,))
x = Dense(10, input_dim=1, activation='relu')(x0)

for i in range(20):

    x_temp=x
    x = Dense(10, activation='relu')(x)
    x = Dense(10, activation='relu')(x)
    x=concatenate([x_temp,x])

x1 = Dense(1)(x)

model =  Model(inputs=x0, outputs=x1)

model.summary()
plot_model(model, show_shapes=True, to_file='model_graph.png')
```

Then, you should compile and fit your model. Results can be plotted after calling the *model.predict* function. As can be seen below, results are good, even with a deep network thanks to residual connexions.

```
# compute model prediction on the validation data
y_p=model.predict(X_val)

# plot the prediction (in red) and the exact values (in blue)
plt.plot(X_val,y_p,'r-')
plt.plot(X_val,y_val,'b-')
plt.title('Prediction and exact data')
```

```
Text(0.5, 1.0, 'Prediction and exact data')
```



<u>Exercice</u>:

Build a neural network that has the structure of the one shown in *model_graph.png* and train it so that it learns the function *y=sin(12πx)*.
Hint: the neural network is the duplication of the same structure four times.
You can use *1000* points for training and *100* points for validating.