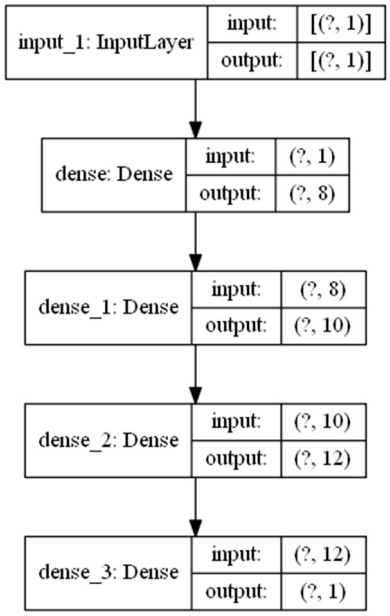
Using Keras Functional API to design complex neural networks (part I)

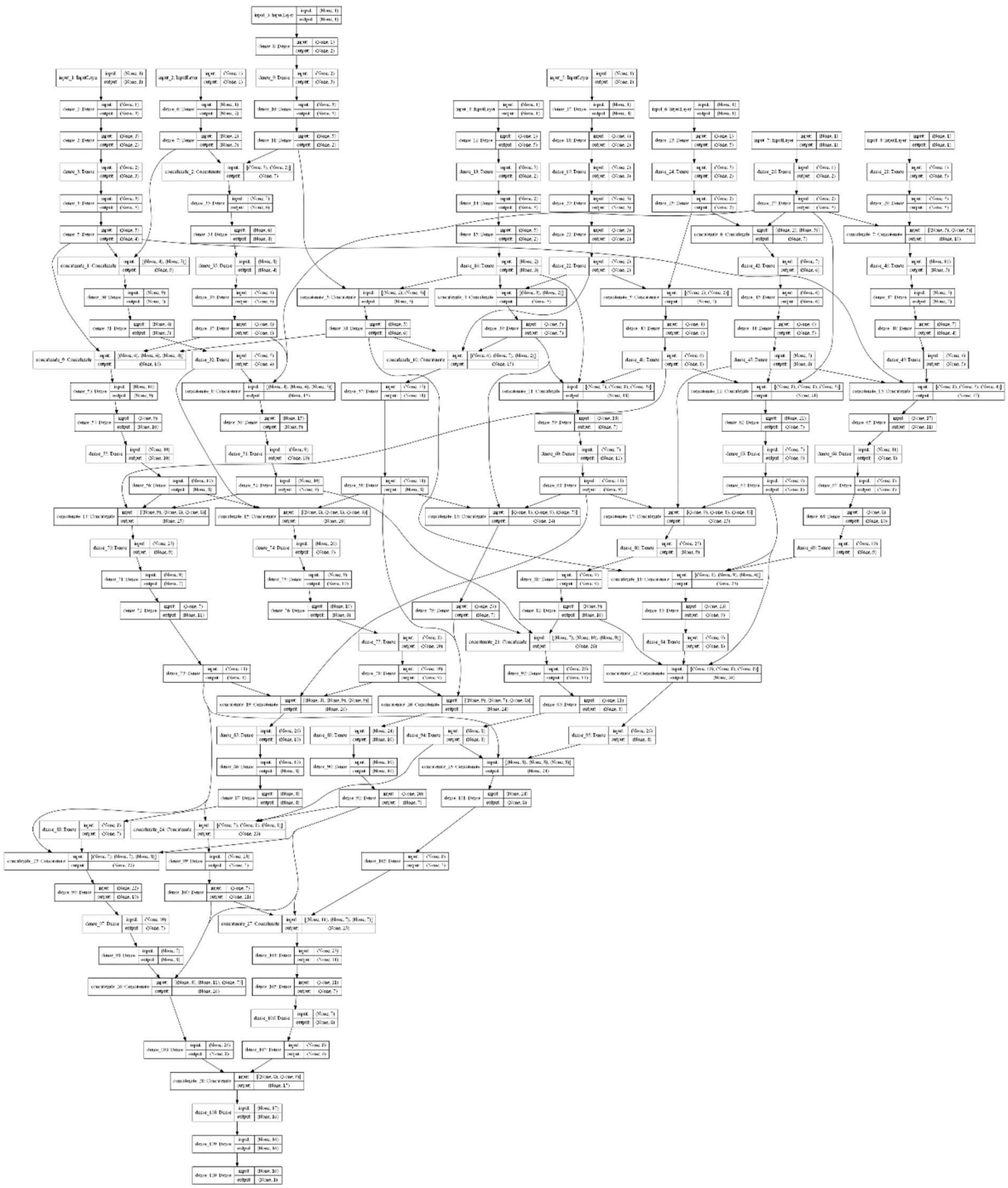
1. Introduction

This tutorial will bring you closer to being able to develop state-of-the-art models on difficult problems using the Keras *functional API* (Application Programming Interface). The Keras *functional API* is a way to create models that are more flexible than the *sequential() API*. The functional API can handle complex models with shared layers and multiple inputs or outputs. The main idea is that a deep learning model is usually a directed acyclic graph (DAG) of layers so the functional API is a way to build *graphs of layers*.

Until now, all neural networks we've seen have been implemented using the Sequential model. The Sequential model makes the assumption that the network has exactly one input and exactly one output, and that it consists of a linear stack of layers as typically shown below for a three hidden layers neural network with 8, 10 and 12 neurons respectively. The question mark means that the first dimension of the matrices can be set to any number.



This is a commonly verified assumption; the configuration is so common that we've been able to cover many topics and practical applications using only the Sequential model class. But this set of assumptions is too inflexible in a number of cases. Some networks require several independent inputs, others require multiple outputs, and some networks have internal branching between layers that makes them look like *graphs* of layers rather than linear stacks of layers. Typical example of such neural network is shown below. Keras functional API is so powerful that this network has been coded in just 25 lines!



2. One input and one output network

First, let's see how to code a neural network with a single input and a single output using Keras functional API. The input and the output can be vectors. What we mean by "one input and one output" is that the entry of the neural network for one sample is a single vector and the output is also a single vector, as opposed to multiple vectors. In this section, we will treat the example of tutorial 1 (the neural network learns the sin function) and see how to code it with Keras functional API. We recall the basic code that enables us to design a neural network that learns the sinus function with the sequential API below:

```
# import librairies
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense

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

# create training and validation data
N=1000
X_train=np.linspace(0.0, 1.0, N)
y_train=np.sin(2*np.pi*X_train)

X_val=np.random.rand(100)
X_val=np.sort(X_val)
y_val=np.sin(2*np.pi*X_val)

# define model
model=Sequential()
model.add(Dense(32, input_dim=1, kernel_initializer='uniform', activation='sigmoid'))
model.add(Dense(32, kernel_initializer='uniform', activation='sigmoid'))
model.add(Dense(1, kernel_initializer='uniform'))

# compile the model and compute the parameters of the model
model.compile(optimizer='rmsprop',loss='mse',metrics=['mse'])
model.fit(X_train,y_train,epochs=400, batch_size=32, validation_data=(X_val,y_val))
```

The only things we need to modify to recode this with functional API is what's below the comment # define model. First, we define one function for each layer with the same parameters as before (function_1, function_2 and function_3 below).

```
function_1 = Dense(32, input_dim=1, kernel_initializer='uniform', activation='sigmoid')
function_2 = Dense(32, kernel_initializer='uniform', activation='sigmoid')
function_3 = Dense(1, kernel_initializer='uniform')
```

Then, the input tensor *x0* will be fed to function_1, giving the output tensor *x1*; this last tensor will then be fed into function_2, and so on.

```
x0 = Input(shape=(1,))
x1 = function_1(x0)
x2 = function_2(x1)
x3 = function_3(x2)
```

The model is then created by simply specifying its input and output:

```
model =  Model(inputs=x0, outputs=x3)
```

The full code is given below, with the new libraries that should be imported to be able to use Keras functional API.

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

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

# create training and validation data
N=1000
X_train=np.linspace(0.0, 1.0, N)
y_train=np.sin(2*np.pi*X_train)

X_val=np.random.rand(100)
X_val=np.sort(X_val)
y_val=np.sin(2*np.pi*X_val)

# define model
function_1 = Dense(32, input_dim=1, kernel_initializer='uniform', activation='sigmoid')
function_2 = Dense(32, kernel_initializer='uniform', activation='sigmoid')
function_3 = Dense(1, kernel_initializer='uniform')

x0 = Input(shape=(1,))
x1 = function_1(x0)
x2 = function_2(x1)
x3 = function_3(x2)

model =  Model(inputs=x0, outputs=x3)

# compile the model and compute the parameters of the model
model.compile(optimizer='rmsprop',loss='mse',metrics=['mse'])
model.fit(X_train,y_train,epochs=400, batch_size=32, validation_data=(X_val,y_val))
```

In general, we do not specify the functions by giving them a name, but we concatenate two instructions of the type

```
function_2 = Dense(32, kernel_initializer='uniform', activation='sigmoid')
x2 = function_2(x1)
```

in only one line as follows:

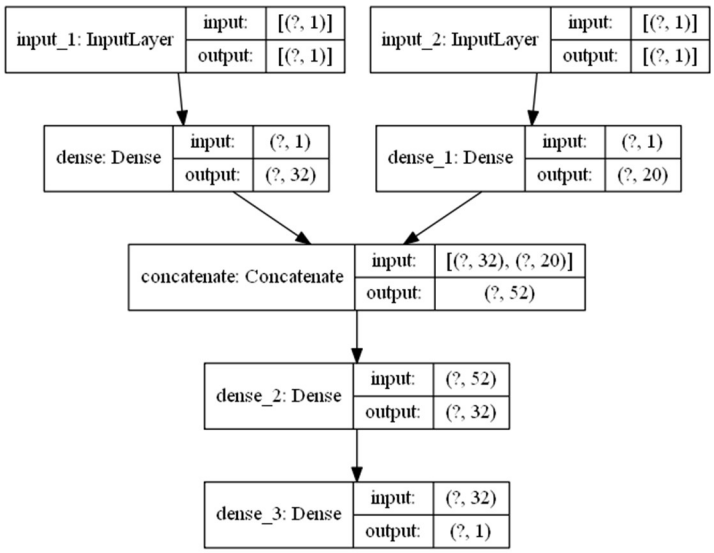
```
x2 = Dense(32, kernel_initializer='uniform', activation='sigmoid')(x1)
```

Doing the same way for all the layers, the model can be coded as follows:

```
# define model
x0 = Input(shape=(1,))
x1 = Dense(32, input_dim=1, kernel_initializer='uniform', activation='sigmoid')(x0)
x2 = Dense(32, kernel_initializer='uniform', activation='sigmoid')(x1)
x3 = Dense(1, kernel_initializer='uniform')(x2)

model =  Model(inputs=x0, outputs=x3)
```

3. Multiple inputs and one output network

The real interest of using functional API relies on its capacity to deal with complex networks. As a first step, assume we want to create a network with two inputs and one output that looks like this one:



Let's see how to build such network steps by steps. First, we must define the two inputs tensors, called *x0_left* and *x0_right* for the left and the right part of the graph, respectively. In this example, we assume that both entries take a scalar, hence *shape=(1,)* but in general, they can have different shapes.

```
x0_left = Input(shape=(1,))
x0_right = Input(shape=(1,))
```

Then, we define the tensors *x1_left* and *x1_right* that result from the dense layer of the left part and the dense layer of the right part, with a number of neurons equal to 32 and 20, respectively.

```
x1_left = Dense(32, kernel_initializer='uniform', activation='sigmoid')(x0_left)
x1_right = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x0_right)
```

Since our neural network contains only one output, at some point, we'll need to merge the two parts of the graph. This can be done in different ways: average, add, subtract, multiply layers. We cannot use these because *x1_left* and *x1_right* do not have the same lengths (32 for the first one and 20 for the second one). Therefore, we use concatenate which results in a tensor of size 32+20=52.

```
x2 = concatenate([x1_left, x1_right])
```

The rest of the model follows the same line as what we have already seen before.

```
x3 = Dense(32, kernel_initializer='uniform', activation='sigmoid')(x2)
x4 = Dense(1, kernel_initializer='uniform')(x3)
```

The compilation of the model is also the same as before. When it comes to fit the model with the data, since there are two inputs, we should feed the input of the neural network with a list of two tensors: one for the left part of the network and another one for the right part of the network. In general, the data given to the inputs of the network are different. But here, for simplicity we will give the same data for the two inputs, hence the argument *[X_train,X_train]* in the *model.fit()* function.

```
model.compile(optimizer='rmsprop',loss='mse',metrics=['mse'])
model.fit([X_train, X_train],y_train,epochs=400, batch_size=32, validation_data=([X_val,X_val],y_val))
```

Assume we want this neural network to learn the sinus function, the complete code is below.

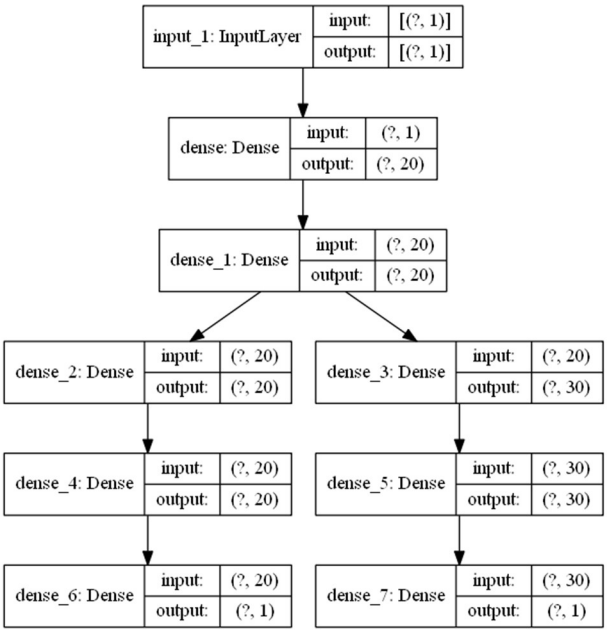
```
# import librairies
import numpy as np
from keras.models import Model
from keras.layers import Dense
from keras import Input
from keras.layers.merge import concatenate

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

# create training and validation data
N=1000
X_train=np.linspace(0.0, 1.0, N)
y_train=np.sin(2*np.pi*X_train)

X_val=np.random.rand(100)
X_val=np.sort(X_val)
y_val=np.sin(2*np.pi*X_val)

# define model
x0_left = Input(shape=(1,))
x0_right = Input(shape=(1,))

x1_left = Dense(32, kernel_initializer='uniform', activation='sigmoid')(x0_left)
x1_right = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x0_right)

x2 = concatenate([x1_left, x1_right])

x3 = Dense(32, kernel_initializer='uniform', activation='sigmoid')(x2)
x4 = Dense(1, kernel_initializer='uniform')(x3)

model =  Model(inputs=[x0_left,x0_right], outputs=x4)

# compile the model and compute the parameters of the model
model.compile(optimizer='rmsprop',loss='mse',metrics=['mse'])
model.fit([X_train, X_train],y_train,epochs=400, batch_size=32, validation_data=([X_val,X_val],y_val))
```

We can think of this network as having two parts that will learn different patterns of the sinus function. Then, we concatenate their respective outputs and add a hidden layer with the hope to obtain a better representation of the sinus function.

Once you have implemented and tested this neural network, you must modify it so that it now has three inputs instead of two.

4. One input and multiple outputs network

Here again, we will implement a toy example, just to see how a multiple output neural network can be built. The network we want to code has one input (a scalar) and two outputs: one that will learn a sinus function and the other one that will learn the exponential function. The graph of the neural network is shown below.



First, we need to create the training data and the validation data to train and test our network:

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

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

(y_train1,y_val1) for the sinus function and (y_train2,y_val2) for the exponential function.

```
# define model
x0 = Input(shape=(1,))

x1 = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x0)
x2 = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x1)

x3_left = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x2)
x3_right = Dense(30, kernel_initializer='uniform', activation='sigmoid')(x2)

x4_left = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x3_left)
x4_right = Dense(30, kernel_initializer='uniform', activation='sigmoid')(x3_right)

x5_left = Dense(1, kernel_initializer='uniform')(x4_left)
x5_right = Dense(1, kernel_initializer='uniform')(x4_right)

model =  Model(inputs=x0, outputs=[x5_left,x5_right])
```

Then, the network can be built in the same way as before (with *x0* the input and *[x5_left, x5_right]* the two outputs.

For the loss function to be used for the training, several options exist: for example, we can use the mean square error for the two outputs (we set *['mse','mse']* as argument when compiling). We can also set a different loss, for example mean square error for one output and mean absolute error for the other output (in that case, we set *['mse','mae']* or the reverse as argument when compiling).

In this case, the total loss function would be:

$$C = \frac{1}{N}\sum_{i=1}^{N}\left(y_i^{(1)} - \widehat{y_i^{(1)}}\right)^2 + \frac{1}{N}\sum_{i=1}^{N}\left|y_i^{(2)} - \widehat{y_i^{(2)}}\right|,$$

where the $y_i^{(1)}$ stands for the output 1 and $y_i^{(2)}$ stands for the output 2.
We can give more weight to one output or the other by specifying *loss_weights* when compiling. If we set *loss_weights=[1,1]* each of the sums will be multiplied by 1. If we want to give more weight to the second sum, then we could set *loss_weights=[0.1,0.9]*, for example. In the code below, we use the same weights for each part of the loss function.

```
# 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_train,[y_train1,y_train2],epochs=300, batch_size=32, validation_data=(X_val,[y_val1,y_val2]))
```

The complete code is shown below.

```
# 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_train=np.linspace(0.0, 1.0, N)
y_train1=np.sin(2*np.pi*X_train)
y_train2=np.exp(X_train)

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

# define model
x0 = Input(shape=(1,))
x1 = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x0)
x2 = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x1)
x3_left = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x2)
x3_right = Dense(30, kernel_initializer='uniform', activation='sigmoid')(x2)
x4_left = Dense(20, kernel_initializer='uniform', activation='sigmoid')(x3_left)
x4_right = Dense(30, kernel_initializer='uniform', activation='sigmoid')(x3_right)
x5_left = Dense(1, kernel_initializer='uniform')(x4_left)
x5_right = Dense(1, kernel_initializer='uniform')(x4_right)

model =  Model(inputs=x0, outputs=[x5_left,x5_right])

# 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_train,[y_train1,y_train2],epochs=300, batch_size=32, validation_data=(X_val,[y_val1,y_val2]))
```

Let's have a look at the different information that Keras output when doing the training (only the output of the last epoch is given below):

```
Epoch 300/300
32/32 [==============================] - 0s 1ms/step - loss: 0.0698 - dense_6_loss: 0.0657 - dense_7_loss: 0.0041 - dense_6_m
se: 0.0657 - dense_7_mse: 0.0041 - val_loss: 0.0589 - val_dense_6_loss: 0.0536 - val_dense_7_loss: 0.0053 - val_dense_6_mse:
0.0536 - val_dense_7_mse: 0.0053
```

The ‘loss’ means the total loss. This loss is made of two contributions: one from the first output (named ‘dense_6’ by Keras) and one from the second output (named ‘dense_7’ by Keras). How do we know the name of the outputs, and more generally the names of the layers that compose the network? Simply by typing the command:

```
plot_model(model,show_shapes=True)
```

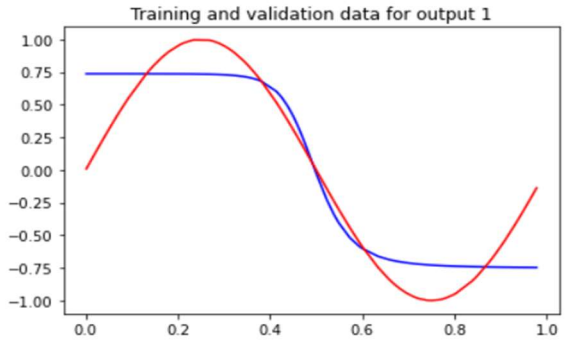
As expected, we have loss = dense_6_loss + dense_7_loss.
Then Keras return the value of dense_6_mse and dense_7_mse that turn out to be the same as dense_6_loss and dense_7_loss, respectively. This is normal since we have chosen the mean square error as a metric for the two outputs. All those losses are computed with the training data. Then we have the same quantities reported with the validation data.

Let’s plot the exact functions versus the function given by the neural network.

```
[y_sin,y_exp]=model.predict(X_val)
```

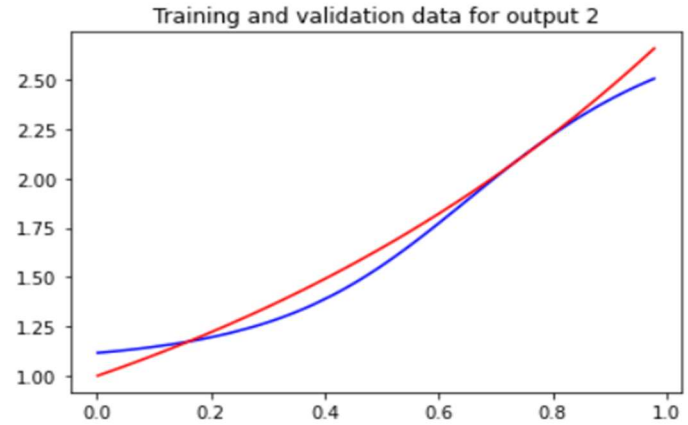
```
plt.plot(X_val,y_sin, 'b')
plt.plot(X_val,y_val1, 'r')
plt.title('Training and validation data for output 1')
```

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




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

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



We can see that the results are not great! It seems that designing a unique network that share its first layers to model a sinus and an exponential function as output is not a good idea. By keeping the same structure (i.e. one input and two outputs) you should try to design a better network.

Exercise:

Build a neural network that have the following structure to learn the sinus function.