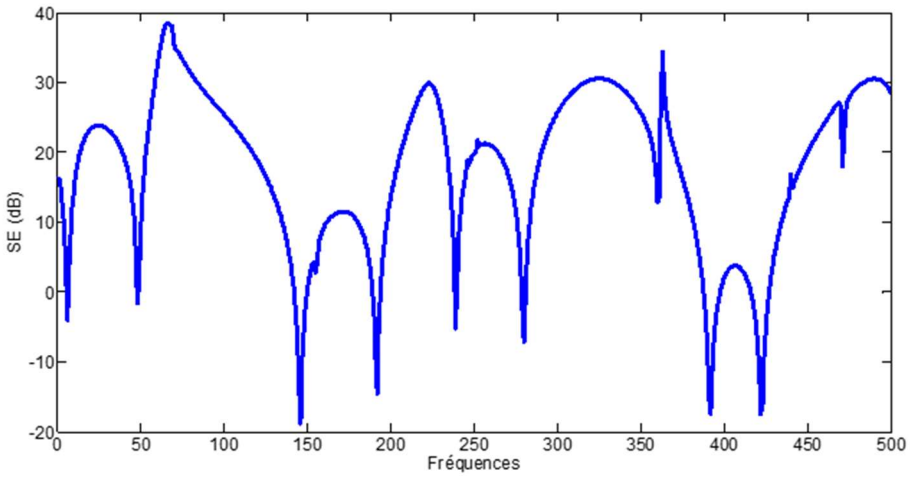
Develop a Neural Network to Model an Electromagnetic Compatibility Problem

1. Tutorial Overview

In this tutorial, we will tackle a more challenging regression problem than just modeling a sinus function. The model we want to build comes from an electromagnetic problem describes below. Assume we have a 3D box with a rectangular opening located at one of its sides. An electromagnetic source is located outside of this box and the emitted waves will enter the box. A sensor (SE) inside the box records the electromagnetic field at one point. We can consider this problem as a shielding problem: the box is assumed to protect some inside equipment from the aggression of electromagnetic waves coming from the outside.

The figure below depicts the problem (plotted in 2D, but the real setting is 3D).

The question we would like to answer is: what are the dimensions of the box (3 parameters), what is the position and the dimensions of the opening (2+2 parameters) and which conductivity of the box (1 parameter) should we choose to obtain the best possible shielding. The electromagnetic field inside the box will be affected by those 8 parameters. Typical electromagnetic field for a given set of 8 parameters is shown below in the frequency domain.



As you can see, the pattern of the curve is much more complex than for the simple sinus we modeled with a neural network. For a given set of parameters, it may take several minutes

to get such results using a scientific computing software. On the other hand, should we be able to model this problem with a neural network, it would only take a fraction of a second to evaluate the model for a given set of new parameters. The search for efficient shielding configuration would then be much easier and faster using a neural network representation of the problem. The size of the input vector of the neural network will be 8 (since we have 8 parameters) and the size of the output vector will be 500 (since the graph is drawn with 500 points). The data set is quite small regarding the dimensions of the input/output spaces as we only have a sample of 1000 input/output $(\vec{x}_i, \vec{y}_i)$ to construct the neural network.

2. Construction of the neural network

2.1 Reading data from a file

We first need to import the libraries that we will use in this tutorial. This is what the following piece of code does:

```
import numpy as np
import time
from keras import models
from keras import layers
from keras import optimizers
import matplotlib.pyplot as plt
```

Then, we need to load the data that will be used to build the neural network. This is done as follows (assuming the file "data1000.txt" is in your current directory):

```
d = np.loadtxt("data_1000.txt")
print(d.shape)
```

```
(508, 1000)
```

It is also a good idea to check the dimensions of the matrix that contains the data. We see that the matrix has 1000 rows and 508 lines. The first 8 lines contain the input data $\vec{x}_i$ and the remaining 500 lines contain the target $\vec{y}_i$. Let's split the data into the training data (X_train,y_train) used to build the model and the testing data (X_test,y_test) used to test the model. 90% are training data and the remaining 10% are test data. Keras expect you to provide the input data under the form of a matrix of size *(Number of data, Number of features)*. It turns out that we need to transpose all the matrix to get the right order. Then, we end up with some sanity checks to ensure that the dimensions of the matrices are correct, by printing them out.

```
X_train=d[0:8,0:900]
X_test=d[0:8,900:]

y_train=d[8:509,0:900]
y_test=d[8:509,900:]

X_train=X_train.T
print('shape of X_train:', X_train.shape)
y_train=y_train.T
print('shape of y_train:', y_train.shape)
X_test=X_test.T
print('shape of X_test:', X_test.shape)
y_test=y_test.T
print('shape of y_test:' ,y_test.shape)
```

```
shape of X_train: (900, 8)
shape of y_train: (900, 500)
shape of X_test: (100, 8)
shape of y_test: (100, 500)
```

2.2 Scaling input variables

Deep learning neural network models learn a mapping from input variables to an output variable. As such, the scale and distribution of the data drawn from the domain may be different for each variable.

Input variables may have different units (e.g. kilograms, meters or hours) that, in turn, may mean the variables have different scales. Differences in the scales across input variables may increase the difficulty of the problem being modelled. An example of this is that large input values (e.g. a spread of hundreds or thousands of units) can result in a model that learns large weight values. A model with large weight values is often unstable, meaning that it may suffer from poor performance during learning and sensitivity to input values resulting in higher generalization error.

A good rule of thumb is that input variables should be small values, probably in the range of 0-1 or standardized with a zero mean and a standard deviation of one. Whether input variables require scaling depends on the specifics of your problem and of each variable. In the previous tutorial we did not need to scale the input since it was already in the range [0,1]. Here it is not the case, and no scaling would prevent the neural network from learning properly. The two main scaling methods replace a feature *x* by *x'* as follows :

$$x' = (x - min)/(max - min)$$

$$x' = (x - mean)/standard\_deviation$$

The first formula ensures that the feature always belongs to the interval *[0,1]* whereas the second one ensures that the feature has zero mean and unit variance. Here, we will use the second formula.

```
mean=X_train.mean(axis=0)
std=X_train.std(axis=0)

# Normalize the features below
```

2.3 Create the Model

This is where your work starts. You must fill the middle part of the code below. You can build upon the neural network given in the previous tutorial. A hint: think big, try 'relu' or 'selu' activation function, do not hesitate to try different things!

```
model=models.Sequential()
# no choice for the input dimension, it must be 8
model.add(layers.Dense(128,activation='relu',input_dim=8))

#
# create your model here
#

# no choice for the output dimension, it must be 500
model.add(layers.Dense(500))
```

2.4 Compile and Fit your Model

Now you have defined your model it is ready to be compiled (see how we did it in the previous tutorial). Then, you can train or fit your model on the data by calling the fit() function on the model. The parameters can be chosen experimentally by trial and error. It is always a good idea to monitor the history of the learning process training loss/validation loss and training error/validation error.

They are stored into the Keras object that we named 'hist' and that contains the dictionary history, as can be seen below. Here we have set *validation_split=0.15* so that Keras uses 15% of the training data as validation data.

```
hist=model.fit(X_train,y_train,epochs=130, batch_size=32, validation_split=0.15)
```

```
print(hist.history.keys())
```

```
dict_keys(['val_loss', 'val_mse', 'loss', 'mse'])
```

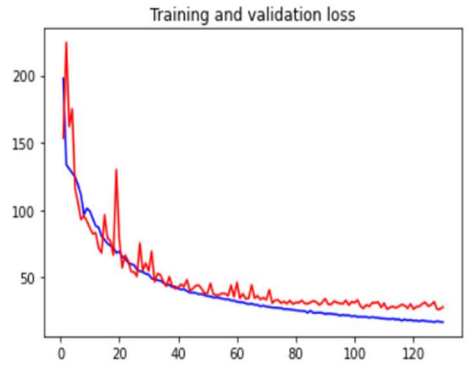
Recall that a dictionary is a sort of array whose entries are called by their keys (here there are four keys 'val_loss' for the validation loss, 'val_mse' for the validation mean square error, 'loss' for the training loss and 'mse' for the training mean square error).

3. Make plot of convergence

We have trained our neural network on the training dataset we can use the information collected in the 'hist' object to create plots as shown below for the training and validation loss as a function of the iterations.

```
# plot of the training and validation loss
train_loss=hist.history['loss']
valid_loss=hist.history['val_loss']
ep=range(1,len(train_loss)+1)
plt.plot(ep,train_loss,'b-')
plt.plot(ep,valid_loss,'r-')
plt.title('Training and validation loss')
```

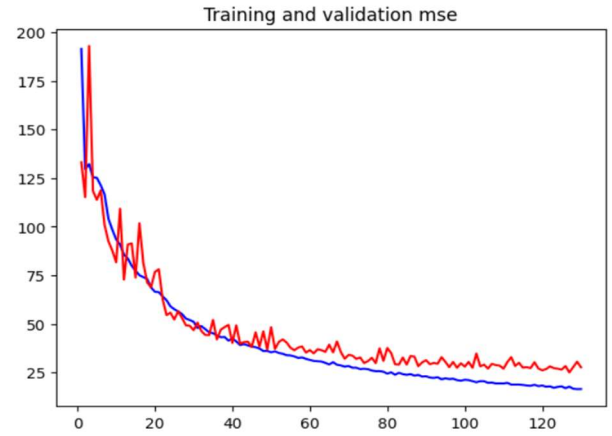
```
Text(0.5, 1.0, 'Training and validation loss')
```



We can proceed in the same way for the training and the validation mean square error.

```
# plot of the training and validation mean square error
train_mse=hist.history['mse']
valid_mse=hist.history['val_mse']
ep=range(1,len(train_loss)+1)
plt.plot(ep,train_mse,'b-')
plt.plot(ep,valid_mse,'r-')
plt.title('Training and validation mse')
```

```
Text(0.5, 1.0, 'Training and validation mse')
```



From those plots, we see that the error stalls after 120 epochs, so it is not necessary to increase the number of iterations.

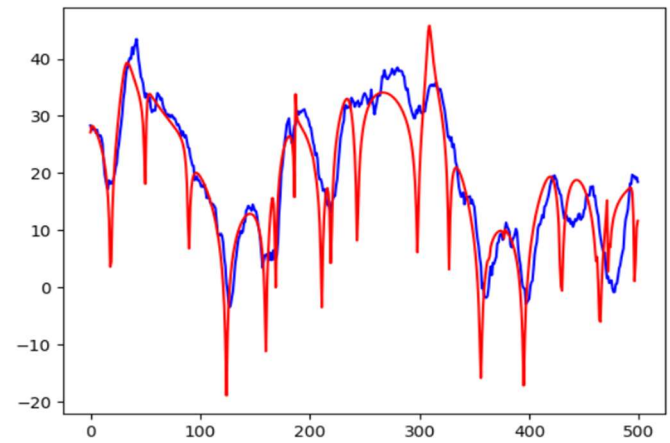
4. Make prediction

How does your model perform on unseen data? You can check that on one example using the predict() Keras function and plot the exact solution vs the solution computed by the model.

```
# make prediction with the model and plot the solution computed by the model
y_p=model.predict(X_test)
plt.plot(y_p[0,:],'b-')
# plot of the exact solution
plt.plot(y_test[0,:],'r-')
```

```
4/4 [==============================] - 0s 3ms/step
```

```
[<matplotlib.lines.Line2D at 0x1c9f18b5e50>]
```



As we can see, it's not as easy as for a sinus function! However, bear in mind that we only used 900 inputs/outputs data to build a model that has an input dimension equal to 8 and an output dimension equal to 500. Considering that, it's not bad at all…

Exercise :

In this example, we have used the mean squared error both for the loss and the metric. For the training, use now the mean absolute error and see if it improves the model (the metric will remain the mean squared error).

Go the Keras web page and choose two other losses that are appropriate for our problem. Test them to see if it improves the model.