Develop Your First Neural Network With Keras

Keras is a powerful and easy-to-use Python library for developing and evaluating deep learning models. It wraps efficient numerical computation libraries such as TensorFlow and allows you to define and train neural network models in a few short lines of code. In this lesson you will discover how to create your first neural network model in Python using Keras. After completing this lesson you will know:

- How to generate training and validation data.
- How to define and compile a simple Neural Network model in Keras.
- How to evaluate a Keras model on a validation dataset.

Let's get started.

1. Tutorial Overview

There is not a lot of code required, but we are going to step over it slowly so that you will know how to create your own models in the future. The steps you are going to cover in this tutorial are as follows:

1. Create Data.
2. Define Model.
3. Compile Model.
4. Fit Model.
5. Evaluate Model.
6. Tie It All Together.

2. Get your neural network learn the sinus function

It can be shown that neural networks can learn any function. No matter what the function, there is guaranteed to be a neural network so that for any possible input, x, the value $f(x)$ (or some close approximation) is output from the network. This result holds even if the function has many inputs, and many outputs (i.e a vector as input and output). So let's check that on a simple example: we'll get the neural network learn the function $f(x)=sin(2\pi x)$, for x in $[0,1]$.

2.1 Create training data and validation data

We first need to generate training data so that the network will learn the sinus function from those data. Here, we use 1000 couples of points (x_i,y_i) with $y_i=sin(2\pi x_i)$. We also need to generate validation data (at different location than the training points) to check that the neural network has properly learnt the sinus function. The validation data consists of 100 couples of points (x_i,y_i) with $y_i=sin(2\pi x_i)$ where the points x_i are generated randomly between 0 and 1. This is what the following piece of code does:

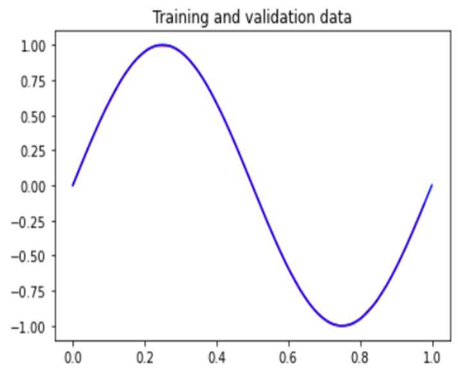
```
import numpy as np

# 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(2*np.pi*X_train)

# generate validation data
X_val=np.random.rand(100)
X_val=np.sort(X_val)
y_val=np.sin(2*np.pi*X_val)
```

Whenever we work with machine learning algorithms that use a stochastic process (e.g. random numbers), it is a good idea to initialize the random number generator with a fixed seed value. This is so that you can run the same code again and again and get the same result. This is useful if you need to demonstrate a result, compare algorithms using the same source of randomness or to debug a part of your code. You can initialize the random number generator with any seed you like, (here *seed=7*, for example).

We can also plot the training data and the validation data. In this example, the two are almost undistinguishable.

```
import matplotlib.pyplot as plt

# plot the validation data in red
plt.plot(X_val,y_val,'r-')

# plot the training data in blue
plt.plot(X_train,y_train,'b-')

plt.title('Training and validation data')
```

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



We have initialized our random number generator to ensure our results are reproducible and we have generated our data. We are now ready to define our neural network model.

2.2 Define Model

Models in Keras are defined as a sequence of layers. We create a Sequential model and add layers one at a time until we are happy with our network topology. The first thing to get right is to ensure the input layer has the right number of inputs. This can be specified when creating the first layer with the input dim argument and setting it to 1 for a single input variable.
How do we know the number of layers to use and their types? This is a very hard question. There are heuristics that we can use and often the best network structure is found through a process of trial and error experimentation. Generally, you need a network large enough to capture the structure of the problem if that helps at all. In this example we will use a fully-connected network structure with three layers.

Fully connected layers are defined using the Dense class. We can specify the number of neurons in the layer as the first argument, the initialization method as the second argument as init and specify the activation function using the activation argument. In this case we initialize the network weights to a small random number generated from a uniform distribution (uniform). Another traditional alternative would be normal for small random numbers generated from a Gaussian distribution.
We will use the sigmoid activation function for all the layers except the last layer which has no activation function since we are performing a regression. The first layer has 32 neurons and expects 1 input variables. The second hidden layer also has 32 neurons and finally the output layer has 1 neuron. The command summary() provides a depiction of the network structure with the total number of parameters to be computed.

```
from keras.models import Sequential
from keras.layers import Dense

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'))

model.summary()
```

```
Model: "sequential_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_47 (Dense)             (None, 32)                64
_________________________________________________________________
dense_48 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_49 (Dense)             (None, 1)                 33
=================================================================
Total params: 1,153
Trainable params: 1,153
Non-trainable params: 0
_________________________________________________________________
```

2.3 Compile Model

Now that the model is defined, we can compile it. When compiling, we must specify some additional properties required when training the network. Remember training a network means finding the best set of weights to make predictions for this problem.
We must specify the loss function used for the computation of weights, the optimizer and any optional metrics we would like to collect and report during training. In this case we will use mean square error loss which is defined in Keras as 'mse'. Recall that the mean square error loss function is defined as

$$C = \frac{1}{N}\sum_{i=0}^{N}(y_i - \widehat{y_i})^2,$$

where $y_i = sin(2\pi x_i)$ is the exact output for an input x_i and $\widehat{y_i}$ is the output of the neural network for an input x_i. And N is the number of training data.

We will also use the efficient gradient descent algorithm 'rmsprop' for no other reason that it is an efficient default. Finally, because it is a regression problem, we will collect and report the mean square error (difference between the exact and the approximate function) as the metric.
Here is the code:

```
model.compile(optimizer='rmsprop',loss='mse',metrics=['mse'])
```

2.4 Fit Model

We have defined our model and compiled it ready for efficient computation. Now it is time to execute the model on some data. We can train or fit our model on our data by calling the fit() function on the model.
The training process will run for a fixed number of iterations through the whole dataset called epochs, that we must specify using the epochs argument. We can also set the number of data used for the evaluation of the gradient, called the batch size by using the batch_size argument. For this problem we will run for a small number of epochs (400) and use a relatively small batch size of 32. Again, these can be ehosen experimentally by trial and error.

```
model.fit(X_train,y_train,epochs=150, batch_size=32, validation_data=(X_val,y_val))
```

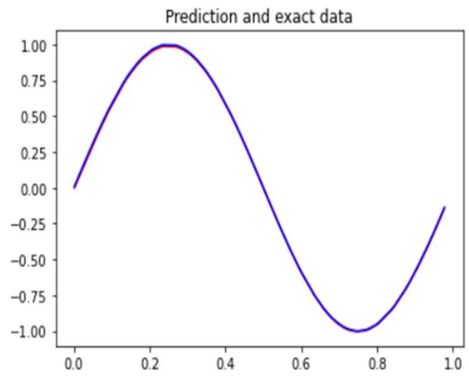
```
Train on 1000 samples, validate on 100 samples
Epoch 1/150
1000/1000 [==============================] - 0s 31us/step - loss: 1.0796e-04 - mse: 1.0796e-04 - val_loss: 2.4825e-04 - val_m
se: 2.4825e-04
Epoch 2/150
1000/1000 [==============================] - 0s 16us/step - loss: 1.0105e-04 - mse: 1.0105e-04 - val_loss: 1.2580e-04 - val_m
se: 1.2580e-04
Epoch 3/150
1000/1000 [==============================] - 0s 16us/step - loss: 1.1377e-04 - mse: 1.1377e-04 - val_loss: 3.0478e-04 - val_m
se: 3.0478e-04
Epoch 4/150
1000/1000 [==============================] - 0s 31us/step - loss: 8.2957e-05 - mse: 8.2957e-05 - val_loss: 3.1501e-04 - val_m
se: 3.1501e-04
Epoch 5/150
1000/1000 [==============================] - 0s 28us/step - loss: 1.1379e-04 - mse: 1.1379e-04 - val_loss: 5.1040e-05 - val_m
se: 5.1040e-05
Epoch 6/150
1000/1000 [==============================] - 0s 22us/step - loss: 1.1380e-04 - mse: 1.1380e-04 - val_loss: 1.5950e-04 - val_m
se: 1.5950e-04
```

2.5 Make predictions with the Model

We have trained our neural network on the training dataset and we could evaluate the performance of the network on the same dataset. However, this is not a good idea since it won't tell us how well the model might perform on new data.
We can make prediction with our model on the validation dataset using the predict() function on the model. This will generate a prediction for each input of the validation set. We can then compare with the exact solution.

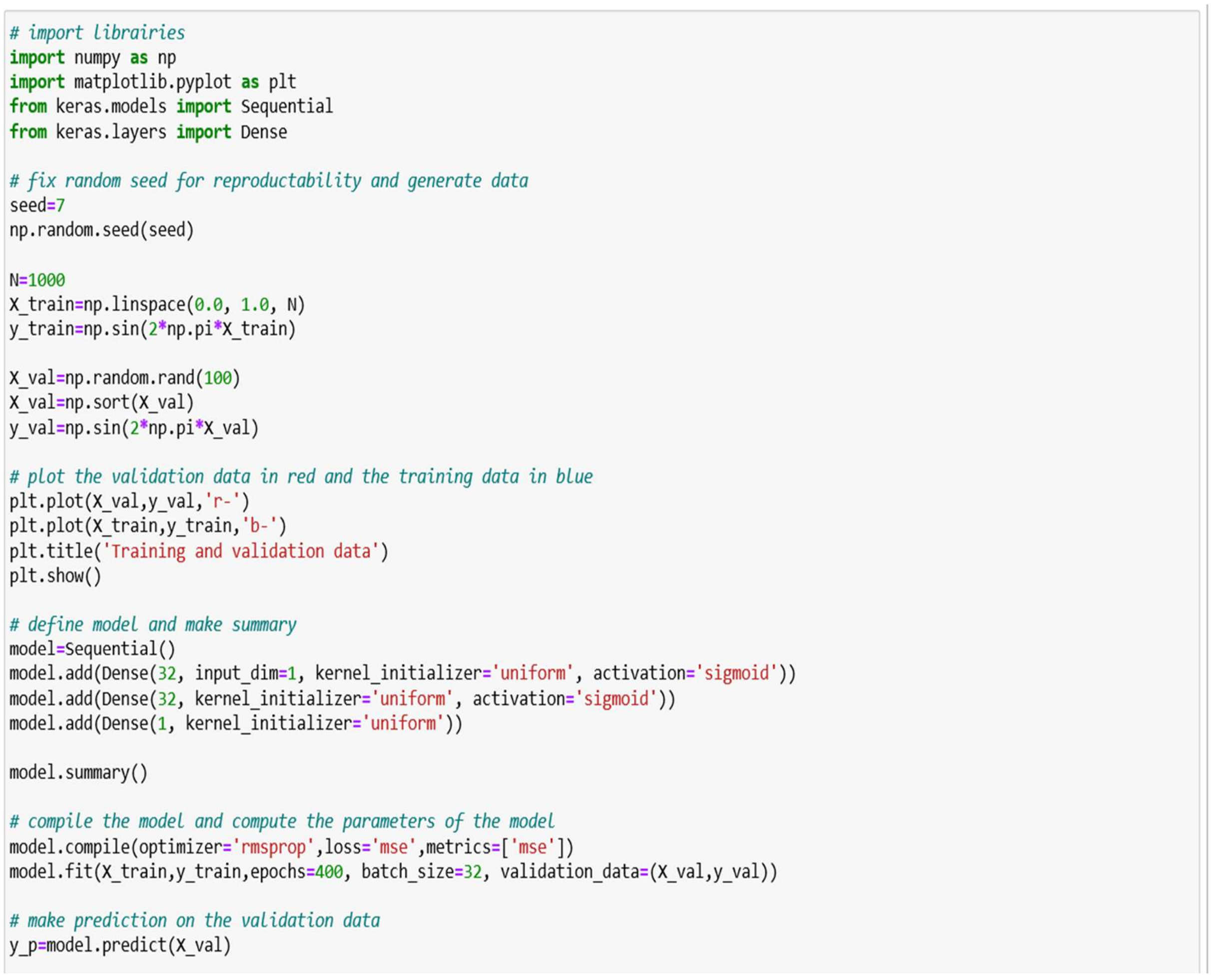
```
# compute model prediction on the validation data
y_p=model.predict(X_val)

# 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')
```



2.6 Tie It All Together

You have just seen how you can easily create your first neural network model in Keras. Let's tie it all together into a complete code example:



```python
# 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)

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)

# plot the validation data in red and the training data in blue
plt.plot(X_val,y_val,'r-')
plt.plot(X_train,y_train,'b-')
plt.title('Training and validation data')
plt.show()

# define model and make summary
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'))

model.summary()

# 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))

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

3. Your work

Exercise 1:

1. Code the given example and modify the number of neurons in the layers to see how it affects the performance of the model on the validation data.
2. What is the mathematical expression of the ReLU activation function? Test this activation function instead of the sigmoid function (you might want to go to the Keras web page to figure out how to utilize this activation function).
3. Is it correct to only have training and validation data? Make a correction to the build a model in accordance with the rules of the arts.
4. Add more layers to your model. Does it improve the performance of the model?
5. The last layer has a linear activation function. In theory, which values can take the output of your model? Since the sinus function takes values in [-1,1], it would be better if the model could only output values within that range. Go to the Keras web page and find an activation function that satisfies this property.

Use this activation function and see if it improves the model.

Exercice 2:

Generate a 100x100 uniform grid on [0,1]x[0,1] (i.e. 10000 points in total). Split your data according to 60%-20%-20% for training, validation and test. Build a neural that learn the function f(x,y)=x^2 +y^2 on [0,1]x[0,1].

Exercice 3:

Build a neural that learn the parametric function $\begin{cases} x(t) = \sin(2\pi t) \\ y(t) = \cos(2\pi t) \end{cases}, t \in [0,1]$ using a dataset of size 1000. To check your model, you can plot the exact parametric curve and the curve built from the data generated by your model.

4. Summary

In this lesson you discovered how to create your first neural network model using the powerful Keras Python library for deep learning. Specifically, you learned the five key steps in using Keras to create a neural network or deep learning model, step-by-step including:

- How to create data.
- How to define a neural network model in Keras.
- How to compile a Keras model.
- How to train a model on data.
- How to evaluate a model on data.

5. Next

In the next tutorial, you will have to create a neural network with data coming from an EMC (Electromagnetic Compatibility) problem.