3 Learning How To Multiply!

For the start, let us build perhaps the *simplest* ANN, one that is tasked to learn how to multiply two numbers between 0 and 1 through observation of examples only! That means, we are going to generate a large number of (few thousand) random float pairs between 0 and 1, multiply them to get the answers (also called "labels"), and then show all of these to a simple fully-connected (FC) neural network we build in `Keras`. Hopefully, our network learns what to do (how to multiply two such numbers). To test that, we generate another, maybe smaller, set of random floats and test the network with them (this can happen during the training), making sure it has not simply memorized all the numbers, and their products, we provided during training.

Follow the algorithm below to develop a `Keras` script for building and training an ANN to do this task. Then examine what the simplest network architecture that can do this job is.

We begin by importing `numpy` as `np` and using `np.random.random(size=...)` to generate the data set of random numbers and products you need for training/testing. We then store them in proper arrays, e.g., `X_train, Y_train, X_test`, and `Y_test`. Here, `X...` and `Y...` hold the input to our neural network (the two numbers) and the desired output (their product), respectively. These arrays do not have to have the same exact shape, of course, but hold an example in each row, i.e., each row in `X...` contains two input numbers, and each element (row) of `Y...` is their corresponding product.

```
import numpy as np

N = 10000

X = np.random.random(size=[N,2])
Y = X[:,0] * X[:,1]

X_train = X[:8*N//10,:]
Y_train = Y[:8*N//10]

X_test = X[8*N//10:,:]
Y_test = Y[8*N//10:]
```

In this example, we are keeping 80% of the data we produce as training data and 20% of it as testing data. Next, we set our network parameters,

```
N_in = 2
```

```
N_hid = 3
N_out = 1
```

where `N_in` is the number of neurons (perceptrons) in the first (input layer), `N_hid` represents the number of neurons in a middle (hidden) layer, and `N_out` is the number of outputs we expect from the network, which is simply one in this case. Next we need to build the ANN model using `Keras` functions:

```
model = Sequential()
model.add(Dense(N_hid, input_dim=N_in, activation='sigmoid'))
model.add(Dense(N_out, activation='sigmoid'))
```

The first command above sets up a "Sequential" ANN, meaning that you feed the network in the input layer and the information propagates forward to the output layer in a sequential manner. The next set of commands specify the FC ("Dense") layers. The last one of these is always the output layer, and you can have zero, one, or more than one, hidden layer(s) in between. Here, our ANN is shallow and really simple with only one hidden layer containing three perceptrons. The sigmoid activation function at the output layer guarantees the output (the desired product of two numbers between 0 and 1) is a number between 0 and 1. You can add the following command,

```
model.summary()
```

which prints out the summary of trainable parameters in the ANN you have constructed. Next comes setting parameters related to the optimization (training), which is done using the `compile` command:

```
model.compile(loss='mean_absolute_error', optimizer='adam', metrics=['accuracy'])
```

Different options for how the *loss function* (the quantity that is minimized during training) is defined, the optimizer, and the way the measure for success is defined can be found in `Keras`'s websites at `https://keras.io/api/losses/`, `https://keras.io/api/optimizers/`, and `https://keras.io/api/metrics/`.

Next, the fit function does the actual optimization/training given the model and the choice of other parameters

```
history = model.fit(X_train, Y_train, validation_data = (X_test,Y_test),
epochs=100, batch_size=100)
```

Here, an *epoch* is when the network has seen all the training data once. We usually perform many epochs in a given training session as the optimization is done stochastically. The `batch_size` is the amount of data seen by the ANN before an adjustment to its parameters is done. Running this command produces information about the training progression, which you can then plot using the variable `history`:

```
import matplotlib.pyplot as plt

plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='lower right')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper right')
plt.show()
```

If everything looks good, the loss has saturated to a minimum value and the accuracy is close to 100%, we can proceed by testing our trained ANN with another newly generated data set:

```
N = 1000
X = np.random.random(size=2*N).reshape(N,2)
Y = X[:,0]*X[:,1]

y_pred = model.predict(X)

plt.plot(Y,y_pred,'o',markersize=2)
plt.plot([0,1],[0,1],'r-',linewidth=3)
plt.xlabel('Actual Product', fontsize=25)
plt.ylabel('Network Prediction', fontsize=25)
plt.show()
```