COMP6685_Lab5a_Keras_XOR

March 2, 2025

1 Lab 1a: Introduction to Deep Learning with Keras

To start understanding artificial neural networks programming, and how these lead to Deep Neural Networks (DNNs), it is good practice to simulate, and understand, one of the simplest (though not trivial, from a learning point of view) problems in machine learning. This is the case of the XOR (eXlusive OR) problem XOR. This a complex, non-linear problem, requiring the use of a Multi-Layer Perceptron (MLP).

This exercise will also introduce the fundamental concepts in Keras to create a neural network model and to carry out training. This will first use a simple Perceptron (input-output layers) and then a Multi-Layer Perceptron (input-hidden-output layers).

The AND Logic Function

Let us start first with the simpler logic AND function. The AND function takes two binary inputs, where each input can assume a value of 0 or 1 (or True/False). These two binary values constitute the input given to a neural network. The network then gives as output a single binary value representing the resultant AND logic combination of the two inputs, which is either 1 (True) or 0 (False). See the table of the possible binary combinations:

- 0 AND 0 = 0
- 1 AND 0 = 0
- 0 AND 1 = 0
- 1 AND 1 = 1

We will now try and implement this function using a network with just 1 layer after the input (no hidden layers). Our network will thus require an input layer with 2 input units, and one output layer with 1 unit.

Let's start our Keras exercise.

Creating the input and output training data

We first import numpy, the python library used for many maths related functions. In our case we will use it to create the array of the input and output training data sets.

After defining our arrays, let's also print the values of the two arrays, to make sure all is working fine.

[1]:
```
# import of numpy to create the input and output data vectors
import numpy as np
```

```
input_data = np.array([[0,0], [0,1], [1,0], [1,1]])
output_data = np.array([[0], [0], [0], [1]])

print (input_data)
print (output_data)
```

```
[[0 0]
 [0 1]
 [1 0]
 [1 1]]
[[0]
 [0]
 [0]
 [1]]
```

Neural network model definition

We now start creating our neural network model, by first importing a few utilities from Keras. We first import the **Sequential** Keras model, which is the default type of models used in the networks studied in our course.

The second line imports the **Dense** layer type, i.e. a fully connected layer used in standard networks such as Perceptrons/MLPs, and the **Activation** functions (e.g. **Sigmoid** and **ReLu**) to be assigend to the hidden and outputs units.

[2]:
```
#!pip3 install tensorflow

# import of keras functions
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
```

```
2025-02-03 12:29:50.193018: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
```

The next step is to create a **Sequential** model. This is the default model for the neural nerworks created today.

[3]:
```
model = Sequential()
```

Subsequently, we need to create our layers. Our simple perceptron has 2 layers: Input and output. The input layer is simply provided by the vector of our input data, and we do not need to create an actual layer.

The output layer is of the *Dense* type where each unit from the input is fully connected to each unit in the output layer. The layer will consist of only **1** unit to decide if the prediction is 0 (False) or 1 (True). Here we will use the **sigmoid** (logistic) function since the output is either 0 or 1.

[4]:
```
# adding 1 layer (the output layer)

model.add(Dense(1, input_dim=2, activation='sigmoid'))
# We always need to specify the number of input dimensions in the first layer␣
 ↪we add
```

```
/usr/local/anaconda3/envs/ml-algo/lib/python3.12/site-
packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

We will now compile the model, specifying the loss function (**mean_squared_error** in this case), the optimiser (we will use **sgd** for Stochastic Gradient Descent) and the **accuracy** metric.

We then plot a summary of our neural network model.

[5]:
```
# compilation and visualisation of the model we have  created

model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 1)	3

Total params: 3 (12.00 B)

Trainable params: 3 (12.00 B)

Non-trainable params: 0 (0.00 B)

Training (Fit) of the network

We are now ready to train our network model. This uses the Keras function **fit**. This function requires some parameters such as the training data (the arrays for the inputs and the corresponding outputs), the number of **epochs** and the level of verbosity of the training details.

[6]:
```
# Trainng (fit) of the model
model.fit(input_data,output_data, epochs=5000, batch_size=4, verbose=2)
```

```
Epoch 4993/5000
1/1 - 0s - 70ms/step - accuracy: 1.0000 - loss: 0.0817
Epoch 4994/5000
1/1 - 0s - 87ms/step - accuracy: 1.0000 - loss: 0.0817
Epoch 4995/5000
1/1 - 0s - 62ms/step - accuracy: 1.0000 - loss: 0.0817
Epoch 4996/5000
1/1 - 0s - 66ms/step - accuracy: 1.0000 - loss: 0.0816
Epoch 4997/5000
1/1 - 0s - 58ms/step - accuracy: 1.0000 - loss: 0.0816
Epoch 4998/5000
1/1 - 0s - 61ms/step - accuracy: 1.0000 - loss: 0.0816
Epoch 4999/5000
1/1 - 0s - 97ms/step - accuracy: 1.0000 - loss: 0.0816
Epoch 5000/5000
1/1 - 0s - 60ms/step - accuracy: 1.0000 - loss: 0.0816
```

[6]:
```
<keras.src.callbacks.history.History at 0x136c725a0>
```

Test of the trained network

At the end, we can also test the performace of the trained model, with the **predict** function.

[7]:
```
# Test of the prediction after training
model.predict(input_data, verbose=1)
```

```
1/1             0s 348ms/step
```

[7]:
```
array([[0.09552547],
       [0.3068933 ],
       [0.27949288],
       [0.61923414]], dtype=float32)
```

You can see from the results (e.g. accuracy) that the network is capable of learning to predict the correct output binary values for the AND problem. Thus a simple Perceptron with no hidden units can learn a simple, learning problem such as AND.

1.1 The XOR Problem

Let us now try the same model architecture on the logic XOR function. The XOR function returns 1 only if its two binary inputs are different, and 0 otherwise. See the table of the possible binary combinations:

- 0 XOR 0 = 0
- 1 XOR 0 = 1
- 0 XOR 1 = 1
- 1 XOR 1 = 0

We will now define the inputs, outputs, and the model exactly as before, then compile and train our model.

[8]:
```
input_data = np.array([[0,0], [0,1], [1,0], [1,1]])
output_data = np.array([[0], [1], [1], [0]])

model = Sequential()
model.add(Dense(1, input_dim=2, activation='sigmoid'))
model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])

model.summary()

# Training (fit) of the model
model.fit(input_data,output_data, epochs=5000, batch_size=4, verbose=2)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 1)	3

Total params: 3 (12.00 B)

Trainable params: 3 (12.00 B)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/5000
1/1 - 1s - 632ms/step - accuracy: 0.5000 - loss: 0.2717
Epoch 2/5000
1/1 - 0s - 75ms/step - accuracy: 0.5000 - loss: 0.2717
Epoch 3/5000
1/1 - 0s - 71ms/step - accuracy: 0.5000 - loss: 0.2716
Epoch 4/5000
1/1 - 0s - 74ms/step - accuracy: 0.5000 - loss: 0.2716
Epoch 5/5000
1/1 - 0s - 74ms/step - accuracy: 0.5000 - loss: 0.2716
Epoch 6/5000
1/1 - 0s - 172ms/step - accuracy: 0.5000 - loss: 0.2715
Epoch 7/5000
1/1 - 0s - 102ms/step - accuracy: 0.5000 - loss: 0.2715
Epoch 8/5000
1/1 - 0s - 170ms/step - accuracy: 0.5000 - loss: 0.2715
Epoch 9/5000
1/1 - 0s - 99ms/step - accuracy: 0.5000 - loss: 0.2714
Epoch 10/5000
```

Let's look at the prediction for the XOR problem.

Has the network learned?

[9]:
```
# Test of the prediction after training
model.predict(input_data, verbose=1)
```

```
1/1             0s 169ms/step
```

[9]:
```
array([[0.4806915],
       [0.535915 ],
       [0.4541395],
       [0.5093042]], dtype=float32)
```

Adding Hidden Layer to create a Multi-Layer Perceptron (MLP)

We can see that our simple network has failed to learn the XOR function even though it was able to learn the AND function. This is because some problems, such as the XOR, are not linearly separable and require at least one hidden layer to be successfuly modeled. In fact, most non-trivial problems of interest are of this class.

Let us now try a more complex network with a hidden layer. The new MLP network will thus require an input layer with 2 input units, one hidden layer with some (minumum 2) units, and one output layer with 1 unit. This type of topology is called Multi-Layer Perceptron (MLP).

The output layer will be the same as before. The hiddent layer will be **Dense**, i.e. where each unit from the input is fully connected to each unit in the hidden layer. This layer will initially have **16** hidden units. We will use the **relu** activation function for the hidden layer neurons.

Finally, we will compile, train, and evaluate our model.

[10]:
```
model = Sequential()

# adding two layers: hidden, then output
model.add(Dense(16, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])

model.summary()

# Training (fit) of the model
model.fit(input_data, output_data, epochs=5000, batch_size=4, verbose=2)
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 16)	48
dense_3 (Dense)	(None, 1)	17

Total params: 65 (260.00 B)

Trainable params: 65 (260.00 B)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/5000
1/1 - 1s - 527ms/step - accuracy: 0.5000 - loss: 0.2810
Epoch 2/5000
1/1 - 0s - 50ms/step - accuracy: 0.2500 - loss: 0.2808
Epoch 3/5000
1/1 - 0s - 56ms/step - accuracy: 0.2500 - loss: 0.2806
Epoch 4/5000
1/1 - 0s - 55ms/step - accuracy: 0.2500 - loss: 0.2804
Epoch 5/5000
1/1 - 0s - 54ms/step - accuracy: 0.2500 - loss: 0.2802
Epoch 6/5000
1/1 - 0s - 57ms/step - accuracy: 0.2500 - loss: 0.2800
Epoch 7/5000
1/1 - 0s - 51ms/step - accuracy: 0.2500 - loss: 0.2798
Epoch 8/5000
1/1 - 0s - 51ms/step - accuracy: 0.2500 - loss: 0.2796
Epoch 9/5000
1/1 - 0s - 50ms/step - accuracy: 0.2500 - loss: 0.2794
Epoch 10/5000
1/1 - 0s - 51ms/step - accuracy: 0.2500 - loss: 0.2792
Epoch 11/5000
1/1 - 0s - 55ms/step - accuracy: 0.2500 - loss: 0.2790
Epoch 12/5000
1/1 - 0s - 51ms/step - accuracy: 0.2500 - loss: 0.2788
Epoch 13/5000
1/1 - 0s - 56ms/step - accuracy: 0.2500 - loss: 0.2786
Epoch 14/5000
1/1 - 0s - 54ms/step - accuracy: 0.2500 - loss: 0.2784
Epoch 15/5000
1/1 - 0s - 51ms/step - accuracy: 0.2500 - loss: 0.2782
Epoch 16/5000
1/1 - 0s - 53ms/step - accuracy: 0.2500 - loss: 0.2780
Epoch 17/5000
1/1 - 0s - 51ms/step - accuracy: 0.2500 - loss: 0.2778
Epoch 18/5000
1/1 - 0s - 51ms/step - accuracy: 0.2500 - loss: 0.2776
Epoch 19/5000
1/1 - 0s - 50ms/step - accuracy: 0.2500 - loss: 0.2774
```

```
Epoch 4988/5000
1/1 - 0s - 153ms/step - accuracy: 1.0000 - loss: 0.0726
Epoch 4989/5000
1/1 - 0s - 104ms/step - accuracy: 1.0000 - loss: 0.0725
Epoch 4990/5000
1/1 - 0s - 137ms/step - accuracy: 1.0000 - loss: 0.0725
Epoch 4991/5000
1/1 - 0s - 304ms/step - accuracy: 1.0000 - loss: 0.0725
Epoch 4992/5000
1/1 - 0s - 105ms/step - accuracy: 1.0000 - loss: 0.0725
Epoch 4993/5000
1/1 - 0s - 133ms/step - accuracy: 1.0000 - loss: 0.0724
Epoch 4994/5000
1/1 - 0s - 128ms/step - accuracy: 1.0000 - loss: 0.0724
Epoch 4995/5000
1/1 - 0s - 188ms/step - accuracy: 1.0000 - loss: 0.0724
Epoch 4996/5000
1/1 - 0s - 179ms/step - accuracy: 1.0000 - loss: 0.0724
Epoch 4997/5000
1/1 - 0s - 101ms/step - accuracy: 1.0000 - loss: 0.0723
Epoch 4998/5000
1/1 - 0s - 169ms/step - accuracy: 1.0000 - loss: 0.0723
Epoch 4999/5000
1/1 - 0s - 134ms/step - accuracy: 1.0000 - loss: 0.0723
Epoch 5000/5000
1/1 - 0s - 219ms/step - accuracy: 1.0000 - loss: 0.0723
```

[10]:
```
<keras.src.callbacks.history.History at 0x138c390a0>
```

Let's test the prediction for the XOR dataset with the additional hidden layer.

[11]:
```
# Test of the prediction after training
model.predict(input_data, verbose=1)
```

```
1/1             0s 239ms/step
```

[11]:
```
array([[0.37860018],
       [0.707333  ],
       [0.8454486 ],
       [0.18990672]], dtype=float32)
```

1.2 Trying different (hyper)parameters

We can carry out additional simulations, by exploring the role of some (hyper)parameters such as the number of hidden nodes, the learning rate, or a different activation function. The code below is exactly the same as the one for the simlation we tried above. However, we will change the learning rate from the default value of 0.01 to 0.1.

[12]:
```
# We need to import optimizers to explicitly set the learning rate
from tensorflow.keras.optimizers import *

model = Sequential()

model.add(Dense(16, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Define the optimizer and set its learning rate
opt = SGD(learning_rate=0.1)

model.compile(loss='mean_squared_error', optimizer=opt, metrics=['accuracy'])

model.fit(input_data,output_data, epochs=5000, batch_size=4, verbose=2)

model.predict(input_data, verbose=1)
```

```
Epoch 1/5000
1/1 - 2s - 2s/step - accuracy: 0.7500 - loss: 0.2355
Epoch 2/5000
1/1 - 0s - 231ms/step - accuracy: 0.7500 - loss: 0.2349
Epoch 3/5000
1/1 - 0s - 134ms/step - accuracy: 0.7500 - loss: 0.2342
Epoch 4/5000
1/1 - 0s - 305ms/step - accuracy: 0.7500 - loss: 0.2336
Epoch 5/5000
1/1 - 0s - 187ms/step - accuracy: 0.7500 - loss: 0.2330
Epoch 6/5000
1/1 - 0s - 227ms/step - accuracy: 0.7500 - loss: 0.2324
Epoch 7/5000
1/1 - 0s - 84ms/step - accuracy: 0.7500 - loss: 0.2319
Epoch 8/5000
1/1 - 0s - 94ms/step - accuracy: 0.7500 - loss: 0.2314
Epoch 9/5000
1/1 - 0s - 158ms/step - accuracy: 0.7500 - loss: 0.2308
Epoch 10/5000
1/1 - 0s - 135ms/step - accuracy: 0.7500 - loss: 0.2303
Epoch 11/5000
1/1 - 0s - 90ms/step - accuracy: 0.7500 - loss: 0.2297
Epoch 12/5000
1/1 - 0s - 106ms/step - accuracy: 0.7500 - loss: 0.2292
Epoch 13/5000
1/1 - 0s - 137ms/step - accuracy: 0.7500 - loss: 0.2287
Epoch 14/5000
1/1 - 0s - 91ms/step - accuracy: 0.7500 - loss: 0.2282
Epoch 15/5000
1/1 - 0s - 90ms/step - accuracy: 0.7500 - loss: 0.2276
```

```
Epoch 4984/5000
1/1 - 0s - 81ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4985/5000
1/1 - 0s - 49ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4986/5000
1/1 - 0s - 50ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4987/5000
1/1 - 0s - 52ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4988/5000
1/1 - 0s - 53ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4989/5000
1/1 - 0s - 80ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4990/5000
1/1 - 0s - 49ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4991/5000
1/1 - 0s - 49ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4992/5000
1/1 - 0s - 53ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4993/5000
1/1 - 0s - 53ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4994/5000
1/1 - 0s - 86ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4995/5000
1/1 - 0s - 105ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4996/5000
1/1 - 0s - 67ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4997/5000
1/1 - 0s - 58ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4998/5000
1/1 - 0s - 54ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 4999/5000
1/1 - 0s - 87ms/step - accuracy: 1.0000 - loss: 0.0013
Epoch 5000/5000
1/1 - 0s - 53ms/step - accuracy: 1.0000 - loss: 0.0013
1/1             0s 80ms/step
```

[12]:
```
array([[0.0612758 ],
       [0.97566134],
       [0.9746712 ],
       [0.01792363]], dtype=float32)
```

We can see the the network now learns faster. The learning rate is one of the most important parameters that needs to be tuned carefully. If the learning rate is too large or too small for the problem, learning can fail.

Change the code above to try different parameters then execute it. Note how your changes affect the network performance.

1.3 Conclusions

This tutorial helped us understand the very basic concept of the implementation of a simple, multi-layer neural network, using the simple, benchmark problem and dataset of XOR. We are now ready to start exploring neural networks with more interesting training problems and datasets.