Exercise 6.3: Neural Networks in Keras

```
Start coding or generate with AI.
```

```
import numpy as np
import matplotlib.pyplot as plt

# See https://keras.io/
# for extennsive documentation
import tensorflow as tf
from tensorflow import keras

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


#For verifying GPU is used when run locally
sess = tf.compat.v1.Session(config=tf.compat.v1.ConfigProto(log_device_placement=True))
```

```
Device mapping:
/job:localhost/replica:0/task:0/device:GPU:0 -> device: 0, name: NVIDIA GeForce GT
```

Let us visit the problem of wine quality prediction previously encountered one final time. After linear regression and a self-made network, we can now explore the comfort provided by the Keras library.

```
# The code snippet below is responsible for downloading the dataset to
# Google. You can directly download the file using the link
# if you work with a local anaconda setup
import os
if not os.path.exists("winequality-white.csv"):
  !wget https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequali


# load all examples from the file
data = np.genfromtxt('winequality-white.csv',delimiter=";",skip_header=1)

print("data:", data.shape)

# Prepare for proper training
np.random.seed(1234) # set seed to get reproducable results
np.random.shuffle(data) # randomly sort examples

# take the first 3000 examples for training
X_train = data[:3000,:11] # all features except last column
y_train = data[:3000,11]  # quality column
```

```
# and the remaining examples for testing
X_test = data[3000:,:11] # all features except last column
y_test = data[3000:,11] # quality column

print("First example:")
print("Features:", X_train[0])
print("Quality:", y_train[0])
```

```
data: (4898, 12)
First example:
Features: [6.100e+00 2.200e-01 4.900e-01 1.500e+00 5.100e-02 1.800e+01 8.700e+01
 9.928e-01 3.300e+00 4.600e-01 9.600e+00]
Quality: 5.0
```

Below is the simple network from exercise 4.1 implemented using Keras. In addition to the network we define the loss function and optimiser.

```
# See: https://keras.io/api/models/sequential/ and
# https://keras.io/api/layers/core_layers/dense/
# We can use the Sequential class to very easiliy
# build a simple architecture
model = Sequential()
# 11 inputs, 20 outputs, relu
model.add(Dense(250, input_dim=11, activation='relu'))
# 20 inputs (automatically detected by Keras), 1 output, linear activation
model.add(Dense(27, input_dim=11, activation='relu'))
model.add(Dense(1, activation='linear'))
```

```
# Set loss function and optimiser algorithm
model.compile(loss='mse',  # mean squared error
              optimizer='adam'# stochastic gradient descent
             )
```

Training and evaluation below

The code below trains the network for 5 epochs using the loss function and optimiser defined above. Each example is individually passed to the network

```
history = model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=5, batch_size=5)
```

```
Epoch 1/5
600/600 [==============================] - 3s 5ms/step - loss: 0.6336 - val_loss:
Epoch 2/5
600/600 [==============================] - 3s 5ms/step - loss: 0.6496 - val_loss:
Epoch 3/5
```

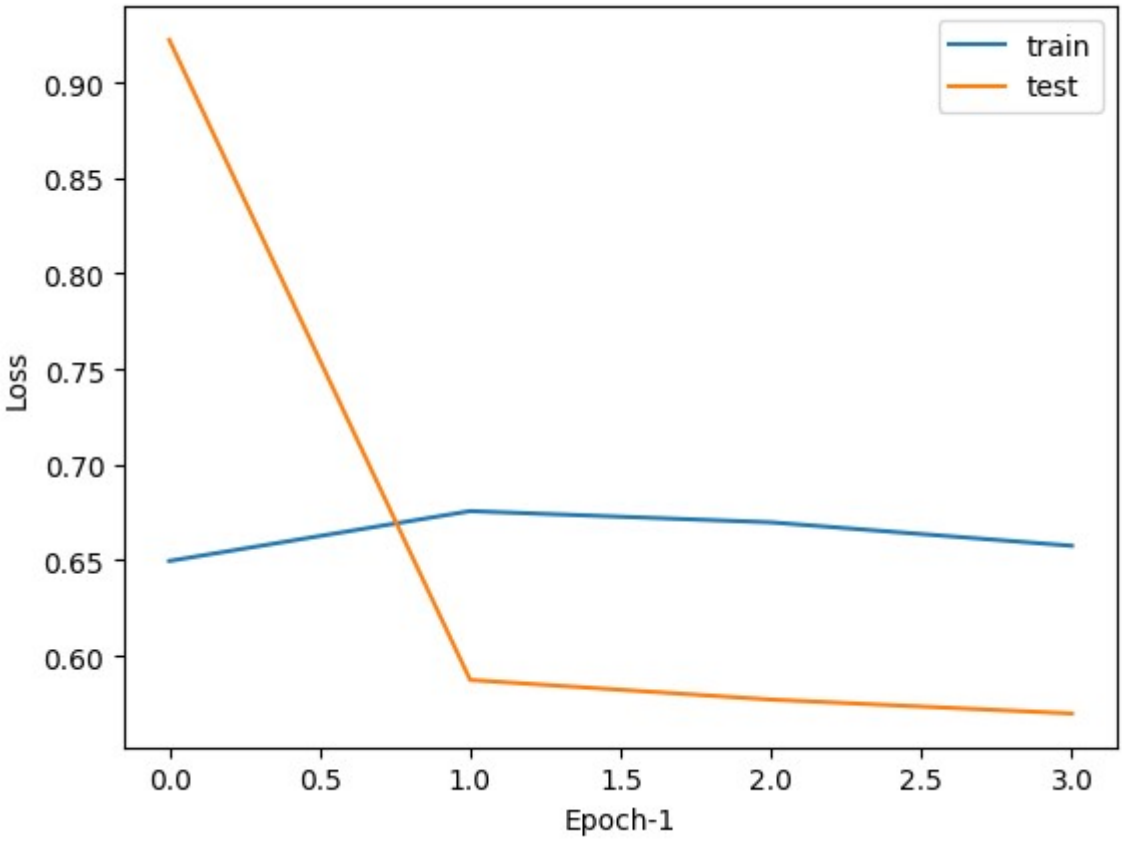
```
Epoch 3/5
600/600 [==============================] - 3s 5ms/step - loss: 0.6757 - val_loss:
Epoch 4/5
600/600 [==============================] - 3s 5ms/step - loss: 0.6698 - val_loss:
Epoch 5/5
600/600 [==============================] - 3s 5ms/step - loss: 0.6576 - val_loss:
```

Note

This is the best I can accomplish by **only changing network architecture**, i.e. not changing the training process. In this case I found reguralization did not help performance.

```
# The history object returned by the model training above
# contains the values of the loss function (the mean-squared-error)
# at different epochs
# We discard the first epoch as the loss value is very high,
# obscuring the rest of the distribution
train_loss = history.history["loss"][1:]
test_loss = history.history["val_loss"][1:]
```

```
# Prepare and plot loss over time
plt.plot(train_loss,label="train")
plt.plot(test_loss,label="test")
plt.legend()
plt.xlabel("Epoch-1")
plt.ylabel("Loss")
plt.show()
```

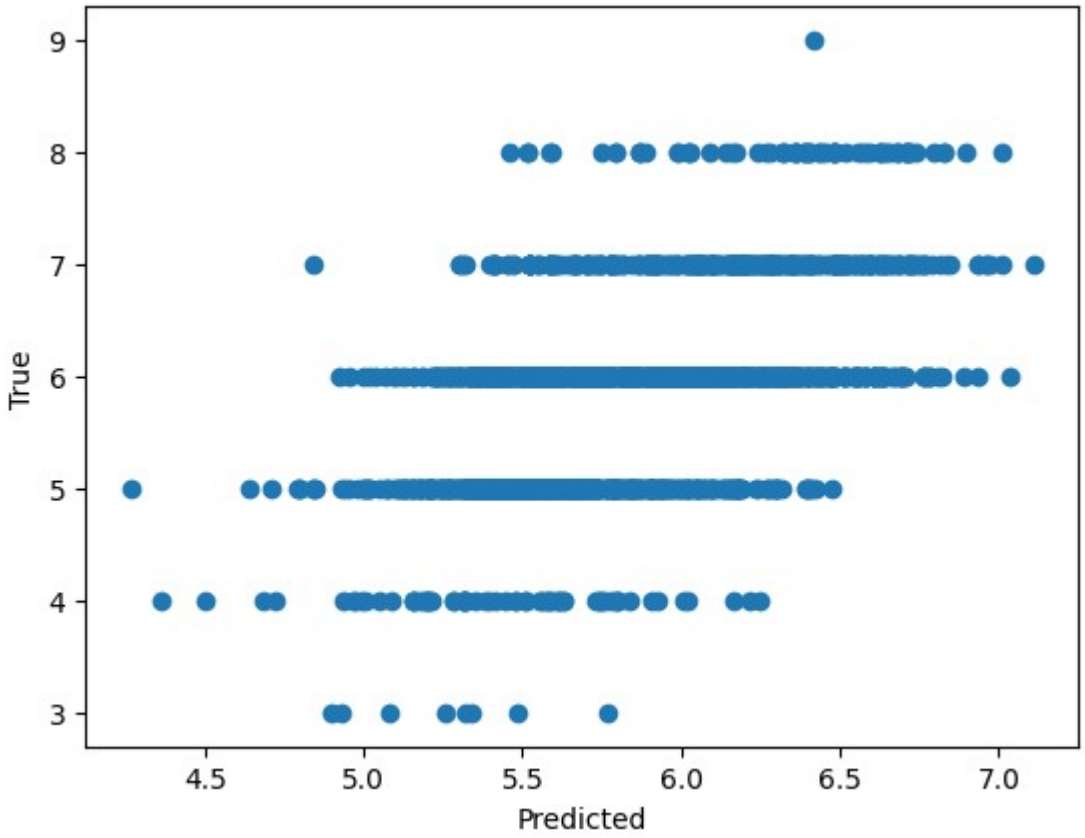


```
# After the training:

# Prepare scatter plot
y_pred = model.predict(X_test)[:,0]

print("Correlation coefficient:", np.corrcoef(y_pred,y_test)[0,1])
plt.scatter(y_pred,y_test)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

```
60/60 [==============================] - 0s 1ms/step
Correlation coefficient: 0.5414947614395764
```




```
np.corrcoef(y_pred,y_test)
```

```
array([[1.        , 0.54149476],
       [0.54149476, 1.        ]])
```

Note

I think this is an acceptable result. Still only just slightly better than linear regression (corrcoef 0.52).

Problems

- Use the notebook as starting point. It already contains the simple network from Exercise 4.1 implemented in Keras.
- Currently, SGD is used without momentum. Try training with a momentum term. Replace SGD with the Adam optimizer and train using that. (See: https://keras.io/api/optimizers/)
- Add two more hidden layers to the network (you can choose the number of nodes but make sure to apply the ReLu activation function after each) and train again.
- Test differet numbers of examples (i.e. change the batch batch size) to be simulataneously used by the network.
- (bonus) optimize the network architecture to get the best correlation coefficient. (Let's see who gets the most out of the data).

Dont mind this:

```
N=500
res2={}
for i in range(N):
    print(i)
    depth=np.random.randint(1, 10)
    width=[]
    dropout=[]
    for j in range(depth):
        width.append(np.random.randint(1,256))
        dropout.append(0.1*np.random.randint(0,7))
    model = Sequential()

    model.add(Dense(width[0], input_dim=11, activation='relu'))
    model.add(keras.layers.Dropout(dropout[0]))

    for j in range(1,depth):
        model.add(Dense(width[j], activation='relu'))
        model.add(keras.layers.Dropout(dropout[j]))

    model.add(Dense(1, activation='linear'))


    # Set loss function and optimiser algorithm
    model.compile(loss='mse',  # mean squared error
                optimizer='adam', metrics=['accuracy']# stochastic gradient descent
                )
    model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=5, batch_size=60, verbose=0)

    loss=model.evaluate(X_test, y_test, verbose=0, batch_size=5, return_dict=False)[0]
    res2[f"{depth}, {width}, {dropout}"]=loss
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
```

```
    57


print(res)
print(min(res.values()))
```

```
    {'9, [19, 136, 48, 82, 75, 93, 198, 167, 74], [0.4, 0.1, 0.4, 0.1, 0.5, 0.30000000
    0.6474922299385071
```

```
print(min(res2.values()))
print(list(res2.keys())[list(res2.values()).index(min(res2.values()))])
```

```
    0.6188573837280273
    2, [255, 27], [0.0, 0.0]
```