# Exercise 5.1

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

**Simple Network**

We continue with the dataset first encountered in the previous exercise. Please refer to the discussion there for an introduction to the data and the learning objective.

Here, we manually implement a simple network architecture

```
# The code snippet below is responsible for downloading the dataset
# - for example when running via Google Colab.
#
# You can also directly download the file using the link if you work
# with a local setup (in that case, ignore the !wget)
import os
if not os.path.exists("winequality-white.csv"):
  !wget https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequa
```

```
# Before working with the data,
# we download and prepare all features

# 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)
np.random.shuffle(data) # randomly sort examples

# take the first 3000 examples for training
# (remember array slicing from last week)
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
```

## Problems

The goal is to implement the training of a neural network with one input layer, one hidden layer, and one output layer using gradient descent. We first (below) define the matrices and initialise with random values. We need W, b, W' and b'. The shapes will be:

- W: (number of hidden nodes, number of inputs) named `W`
- b: (number of hidden nodes) named `b`
- W': (number of hidden nodes) named `Wp`
- b': (one) named `bp`

Your tasks are:

- Implement a forward pass of the network as `dnn` (see below)
- Implement a function that uses one data point to update the weights using gradient descent. You can follow the `update_weights` skeleton below
- Now you can use the code below (training loop and evaluation) to train the network for multiple data points and even over several epochs. Try to find a set of hyperparameters (number of nodes in the hidden layer, learning rate, number of training epochs) that gives stable results. What is the best result (as measured by the loss on the training sample) you can get?

```
# Initialise weights with suitable random distributions
hidden_nodes = 11 # number of nodes in the hidden layer
n_inputs = 11 # input features in the dataset

W = np.random.randn(hidden_nodes,11)*np.sqrt(2./n_inputs)
b = np.random.randn(hidden_nodes)*np.sqrt(2./n_inputs)
Wp = np.random.randn(hidden_nodes)*np.sqrt(2./hidden_nodes)
bp = np.random.randn((1))

print(W.shape)
```

```
(11, 11)
```

```
# You can use this implementation of the ReLu activation function
def relu(x):
    return np.maximum(x, 0)
```

```
def dnn(x,W,b,Wp,bp):
    return Wp@relu(W@x+b)+bp
```

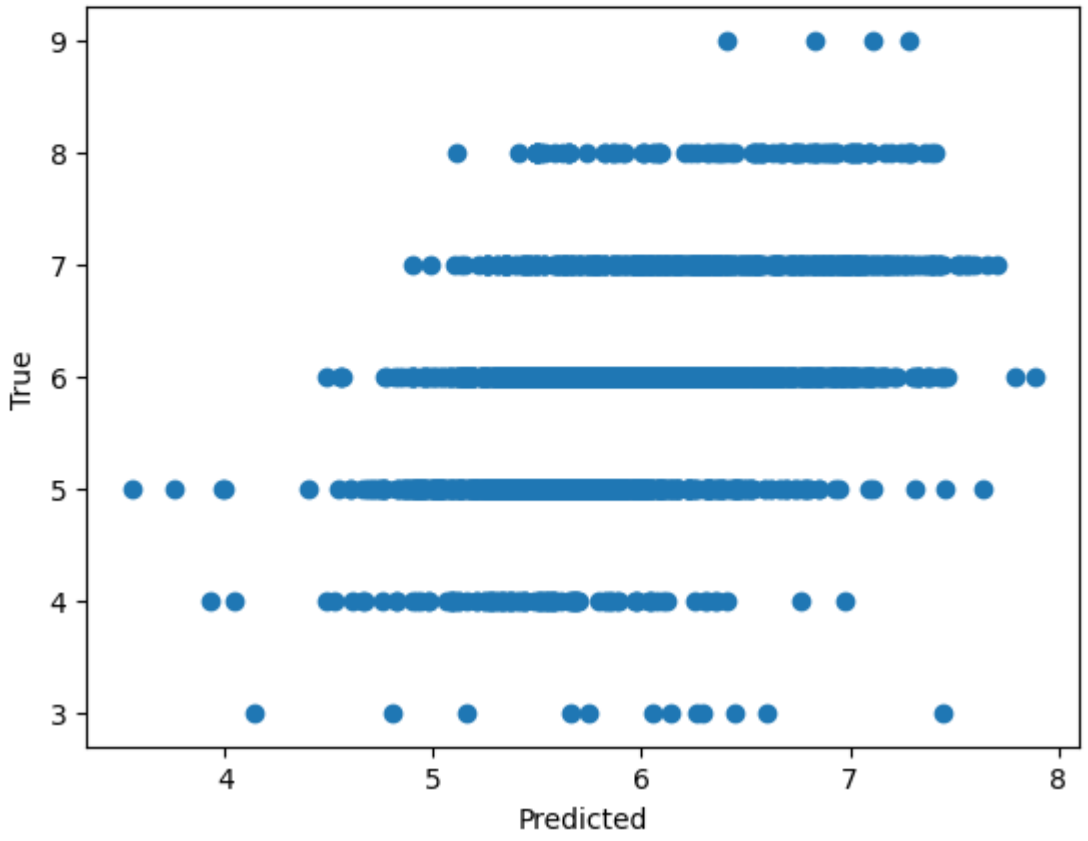
```
def update_weights(x,y, W, b, Wp, bp):

    learning_rate =  1e-5

    yhat=dnn(x,W,b,Wp,bp)

    # TODO: Derive the gradient for each of W,b,Wp,bp by taking the partial
    # derivative of the loss function with respect to the variable and
    # then implement the resulting weight-update procedure
    # See Hint 2 for additional information

    gradbp=2*(yhat-y)
    gradb=2*(yhat-y)*(Wp*(np.heaviside(W@x+b,0)))
    gradWp=2*(yhat-y)*relu(W@x+b)
    gradW=2*(yhat-y)*(np.outer((Wp*np.heaviside(W@x+b,0)),x))

    # You might need these numpy functions:
    # np.dot, np.outer, np.heaviside
    # Hint: Use .shape and print statements to make sure all operations
    # do what you want them to

    # TODO: Update the weights/bias following the rule:  weight_new = weight_old - lea
    bp_new=bp-learning_rate*gradbp
    b_new=b-learning_rate*gradb
    Wp_new=Wp-learning_rate*gradWp
    W_new=W-learning_rate*gradW

    return W_new, b_new, Wp_new, bp_new # return the new weights
```

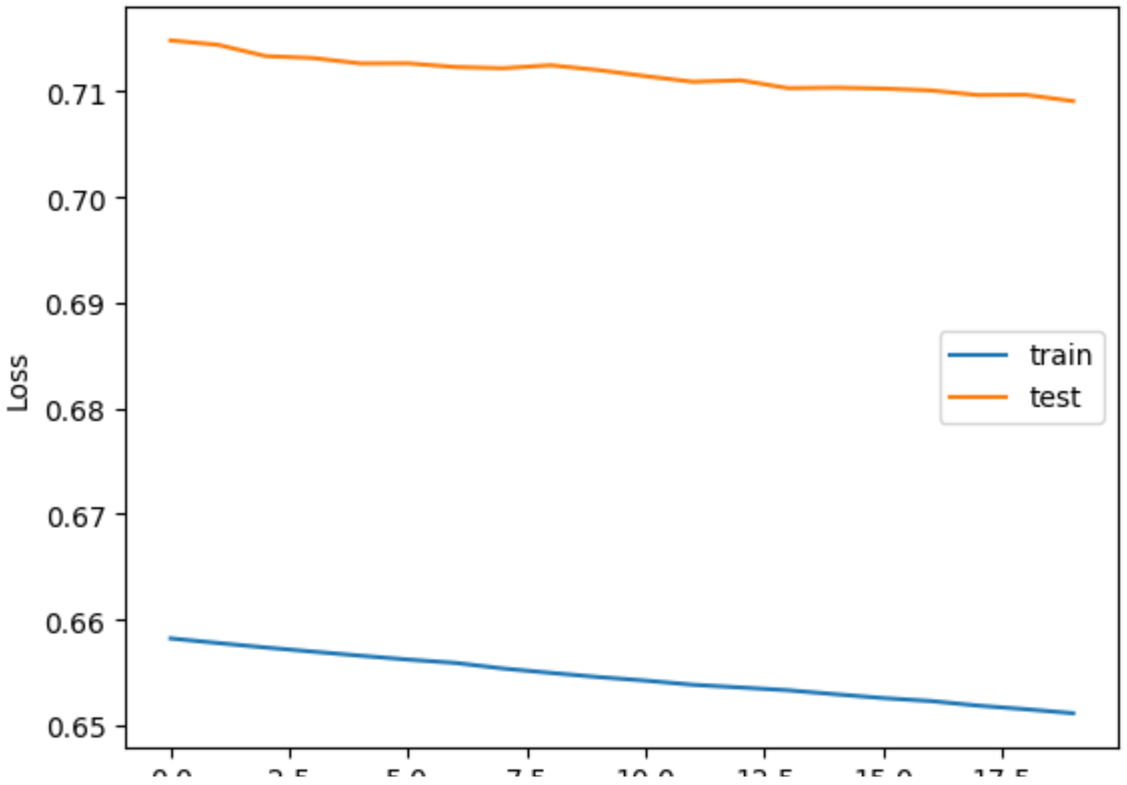
## Training loop and evaluation below

```
# The code below implements the training.
# If you correctly implement  dnn and update_weights above,
# you should not need to change anything below.
# (apart from increasing the number of epochs)

train_losses = []
test_losses = []

# How many epochs to train
# This will just train for one epoch
# You will want a higher number once everything works
n_epochs = 20

# Loop over the epochs
for ep in range(n_epochs):

    # Each epoch is a complete over the training data
    for i in range(X_train.shape[0]):

        # pick one example
        x = X_train[i]
        y = y_train[i]

        # use it to update the weights
        W,b,Wp,bp = update_weights(x,y,W,b,Wp,bp)

    # Calculate predictions for the full training and testing sample
    y_pred_train = [dnn(x,W,b,Wp,bp)[0] for x in X_train]
    y_pred = [dnn(x,W,b,Wp,bp)[0] for x in X_test]

    # Calculate aver loss / example over the epoch
    train_loss = sum((y_pred_train-y_train)**2) / y_train.shape[0]
    test_loss = sum((y_pred-y_test)**2) / y_test.shape[0]

    # print some information
    print("Epoch:",ep, "Train Loss:", train_loss, "Test Loss:", test_loss)

    # and store the losses for later use
    train_losses.append(train_loss)
    test_losses.append(test_loss)


# After the training:

# Prepare scatter plot
y_pred = [dnn(x,W,b,Wp,bp)[0] for x in X_test]

print("Best loss:", min(test_losses), "Final loss:", test_losses[-1])

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

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

```
Epoch: 0 Train Loss: 0.658237413467082 Test Loss: 0.7148430337093407
Epoch: 1 Train Loss: 0.6578000638459827 Test Loss: 0.7144251761343023
Epoch: 2 Train Loss: 0.6573763690622298 Test Loss: 0.7133603829350749
Epoch: 3 Train Loss: 0.656984781275574 Test Loss: 0.7131732795607576
Epoch: 4 Train Loss: 0.6565990951804186 Test Loss: 0.7126604243119354
Epoch: 5 Train Loss: 0.6562338375380323 Test Loss: 0.7126712572663331
Epoch: 6 Train Loss: 0.6559163103607447 Test Loss: 0.712323976029708
Epoch: 7 Train Loss: 0.6553886349723231 Test Loss: 0.7122100776399479
Epoch: 8 Train Loss: 0.6549765822459538 Test Loss: 0.7124913892528221
Epoch: 9 Train Loss: 0.6545806378961576 Test Loss: 0.7120372372896301
Epoch: 10 Train Loss: 0.6542430176577912 Test Loss: 0.7114589699502807
```

```
Epoch: 11 Train Loss: 0.6538475633317238 Test Loss: 0.7109253058250091
Epoch: 12 Train Loss: 0.6535919336333244 Test Loss: 0.7110669009459909
Epoch: 13 Train Loss: 0.6533287739596212 Test Loss: 0.71032371907831
Epoch: 14 Train Loss: 0.6529512731709818 Test Loss: 0.7103737145416046
Epoch: 15 Train Loss: 0.6525901564101144 Test Loss: 0.7102852756132483
Epoch: 16 Train Loss: 0.6523109040962307 Test Loss: 0.710110249182448
Epoch: 17 Train Loss: 0.6518747450138551 Test Loss: 0.709669299644076
Epoch: 18 Train Loss: 0.6515346780338642 Test Loss: 0.7096942079953181
Epoch: 19 Train Loss: 0.6511540925452327 Test Loss: 0.7090979837083302
Best loss: 0.7090979837083302 Final loss: 0.7090979837083302
Correlation coefficient: 0.4297119272639237
```





```
loss={50:0.616,150:1.64,20:0.615,11:0.597}
#Takes quite a few epochs but this is the best i can manage from adaptively changing the
#The graphs are not for a fully optimized network
```

## Hint 1

We want a network with one hidden layer. As activiation in the hidden layer $\sigma$ we apply element-wise ReLu, while no activation is used for the output layer. The forward pass of the network then reads:

$$\hat{y} = \mathbf{W}'\sigma(\mathbf{W}\vec{x} + \vec{b}) + b'$$

## Hint 2

For the regression problem the objective function is the mean squared error between the prediction and the true label $y$:

$$L = (\hat{y} - y)^2$$

Taking the partial derivatives - and diligently the applying chain rule - with respect to the different objects yields:

$$\begin{aligned}
\frac{\partial L}{\partial b'} &= 2(\hat{y} - y) \\
\frac{\partial L}{\partial b_k} &= 2(\hat{y} - y)\mathbf{W}'_k \theta\left(\sum_i \mathbf{W}_{ki} x_i + b_k\right) \\
\frac{\partial L}{\partial \mathbf{W}'_k} &= 2(\hat{y} - y)\sigma\left(\sum_i \mathbf{W}_{ki} x_i + b_k\right) \\
\frac{\partial L}{\partial \mathbf{W}_{km}} &= 2(\hat{y} - y)\mathbf{W}'_m \theta\left(\sum_i \mathbf{W}_{mi} x_i + b_m\right) x_k
\end{aligned}$$

Here, $\Theta$ denotes the Heaviside step function.

## Comments

We need a low ($< 10^{-4}$) learning rate, otherwise we end up with all parameters except for $b'$ as 0. For performance one can start out with a few epochs with a higher learning rate and when loss graph becomes jagged learning rate can be decreased to really find the minimum when loss graph becomes jagged learning rate can be decreased to really find the minimum. After trying a few values for number of hidden nodes the best performance is found for 11 nodes with a loss of 0.597. This is comparable to the linear regression, but still this shallow network performs poorly.