```python
import autograd.numpy as np  # We need to use this numpy wrapper to make automatic differentiation work later
from autograd import grad, elementwise_grad
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

# Defining some activation functions
def ReLU(z):
    return np.where(z > 0, z, 0)

# Derivative of the ReLU function
def ReLU_der(z):
    return np.where(z > 0, 1, 0)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def mse(predict, target):
    return np.mean((predict - target) ** 2)
```

Exercise 2a)

The shape of weights and biases will be determined based on the input and output sizes of the network. In this case, the input size will be 2, and the output size will be 3.

```python
# Exercise 2b)
def feed_forward_one_layer(W, b, x):
    z = W @ x + b
    a = sigmoid(z)
    return a

def cost_one_layer(W, b, x, target):
    predict = feed_forward_one_layer(W, b, x)
    return mse(predict, target)


x = np.random.rand(2)
target = np.random.rand(3)

W = np.random.randn(len(target), len(x))
b = np.random.randn(len(target))

# Exercise 2c)
autograd_one_layer = grad(cost_one_layer, [0, 1])
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g, b_g)
```

```
[[-0.0806643  -0.04842969]
 [-0.0281288  -0.01688813]
 [-0.04492729 -0.0269737 ]] [-0.08157134 -0.0284451  -0.04543248]
```

Exercise 3a)

The reusable results are dC/da and da/dz.

```
# Exercise 3b)
z = W @ x + b
a = sigmoid(z)

predict = a

def mse_der(predict, target):
    return 2/len(predict) * (predict - target)

print(mse_der(predict, target))

cost_autograd = grad(mse, 0)
print(cost_autograd(predict, target))
```

```
[-0.3264923  -0.13291599 -0.49424357]
[-0.3264923  -0.13291599 -0.49424357]
```

```
# Exercise 3c)
def sigmoid_der(z):
    return sigmoid(z) * (1 - sigmoid(z))

print(sigmoid_der(z))

sigmoid_autograd = elementwise_grad(sigmoid, 0)
print(sigmoid_autograd(z))
```

```
[0.24984155 0.2140081  0.09192326]
[0.24984155 0.2140081  0.09192326]
```

```
# Exercise 3d)
dC_da = mse_der(a, target)
dC_dz = dC_da * sigmoid_der(z)

print(dC_da.shape, dC_dz.shape)
print(sigmoid_der(z).shape)
```

```
(3,) (3,)
(3,)
```

```
# Exercise 3e)
dz_dW = np.tensordot(np.eye(len(target)), x, axes=0)
dz_db = np.ones(len(b))
```

```python
# Exercise 3f)
dC_da = mse_der(a, target)
dC_dz = dC_da * sigmoid_der(z)
dC_dW = dC_dz @ dz_dW
dC_db = dC_dz * dz_db

print(dC_dW, dC_db)
```

```
[[-0.0806643  -0.04842969]
 [-0.0281288  -0.01688813]
 [-0.04492729 -0.0269737 ]] [-0.08157134 -0.0284451  -0.04543248]
```

```python
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g, b_g)
```

```
[[-0.0806643  -0.04842969]
 [-0.0281288  -0.01688813]
 [-0.04492729 -0.0269737 ]] [-0.08157134 -0.0284451  -0.04543248]
```

```python
x = np.random.rand(2)
target = np.random.rand(4)

W1 = np.random.rand(3, 2)
b1 = np.random.rand(3)

W2 = np.random.rand(4, 3)
b2 = np.random.rand(4)

layers = [(W1, b1), (W2, b2)]

z1 = W1 @ x + b1
a1 = sigmoid(z1)
z2 = W2 @ a1 + b2
a2 = sigmoid(z2)

# Exercise 4a)
dC_da2 = mse_der(a2, target) # OK
dC_dz2 = dC_da2 * sigmoid_der(z2) # check vector as exponent
dC_dW2 = dC_dz2 @ np.tensordot(np.eye(len(z2)), a1, axes=0)
dC_db2 = dC_dz2 # deriv wrt b2 is 1
```

Exercise 4b)

The derivative of the second layer intermediate z2 wrt. the first layer activation a1 is a row vector where each entry is the sum of the corresponding row in the matrix.

```python
# Exercise 4c)
dC_da1 = dC_dz2 @ W2 # OK
dC_dz1 = dC_da1 * sigmoid_der(z1) # check vector as exponent
dC_dW1 = dC_dz1 @ np.tensordot(np.eye(len(z1)), x, axes=0) # OK
dC_db1 = dC_dz1 # deriv wrt b1 is 1
```

```python
print(dC_dW1, dC_db1)
print(dC_dW2, dC_db2)
```

```
[[0.00412035 0.00806305]
 [0.00421007 0.00823863]
 [0.00358914 0.00702353]] [0.00812245 0.00829932 0.00707527]
[[-0.01723865 -0.02157461 -0.02236588]
 [ 0.03704805  0.04636658  0.04806713]
 [ 0.02956084  0.03699614  0.03835302]
 [ 0.00109094  0.00136534  0.00141542]] [-0.02825852  0.06073115
0.04845771  0.00178833]
```

```python
# Exercise 4d)
def feed_forward_two_layers(layers, x):
    W1, b1 = layers[0]
    z1 = W1 @ x + b1
    a1 = sigmoid(z1)

    W2, b2 = layers[1]
    z2 = W2 @ a1 + b2
    a2 = sigmoid(z2)

    return a2

def cost_two_layers(layers, x, target):
    predict = feed_forward_two_layers(layers, x)
    return mse(predict, target)


grad_two_layers = grad(cost_two_layers, 0)
grad_two_layers(layers, x, target)
```

```
[(array([[0.00412035, 0.00806305],
         [0.00421007, 0.00823863],
         [0.00358914, 0.00702353]]),
  array([0.00812245, 0.00829932, 0.00707527])),
 (array([[-0.01723865, -0.02157461, -0.02236588],
         [ 0.03704805,  0.04636658,  0.04806713],
         [ 0.02956084,  0.03699614,  0.03835302],
         [ 0.00109094,  0.00136534,  0.00141542]]),
  array([-0.02825852,  0.06073115,  0.04845771,  0.00178833]))]
```

Exercise 4e)

The first derivative (the cost function) will be used one time on the outer layer. On the layer in question, we differentiate wrt W or b, but for intermediate layers we differentiate the activation functions and application of weight and bias over and over, until we reach the layer we are interested in.

```python
def create_layers(network_input_size, layer_output_sizes):
    layers = []

    i_size = network_input_size
    for layer_output_size in layer_output_sizes:
        W = np.random.randn(layer_output_size, i_size)
        b = np.random.randn(layer_output_size)
        layers.append((W, b))

        i_size = layer_output_size
    return layers


def feed_forward(input, layers, activation_funcs):
    a = input
    for (W, b), activation_func in zip(layers, activation_funcs):
        z = W @ a + b
        a = activation_func(z)
    return a

def cost(layers, input, activation_funcs, target):
    predict = feed_forward(input, layers, activation_funcs)
    return mse(predict, target)

def feed_forward_saver(input, layers, activation_funcs):
    layer_inputs = []
    zs = []
    a = input
    for (W, b), activation_func in zip(layers, activation_funcs):
        layer_inputs.append(a)
        z = W @ a + b
        a = activation_func(z)

        zs.append(z)

    return layer_inputs, zs, a

# Exercise 5a)
def backpropagation(
    input, layers, activation_funcs, target, activation_ders,
cost_der=mse_der
):
    layer_inputs, zs, predict = feed_forward_saver(input, layers,
activation_funcs)

    layer_grads = [() for layer in layers]

    # We loop over the layers, from the last to the first
    for i in reversed(range(len(layers))):
        layer_input, z, activation_der = layer_inputs[i], zs[i],
```

```
activation_ders[i]

        if i == len(layers) - 1:
            # For last layer we use cost derivative as dC_da(L) can be
computed directly
            dC_da = cost_der(predict, target)
        else:
            # For other layers we build on previous z derivative, as
dC_da(i) = dC_dz(i+1) * dz(i+1)_da(i)
            (W, b) = layers[i + 1]
            dC_da = dC_dz @ W

        dC_dz = dC_da * activation_der(z)
        dC_dW = dC_dz @ np.tensordot(np.eye(len(z)), layer_input,
axes=0)
        dC_db = dC_dz # deriv wrt b is 1

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads

network_input_size = 2
layer_output_sizes = [3, 4]
activation_funcs = [sigmoid, ReLU]
activation_ders = [sigmoid_der, ReLU_der]

layers = create_layers(network_input_size, layer_output_sizes)

x = np.random.rand(network_input_size)
target = np.random.rand(4)

layer_grads = backpropagation(x, layers, activation_funcs, target,
activation_ders)
print(layer_grads)

cost_grad = grad(cost, 0)
cost_grad(layers, x, [sigmoid, ReLU], target)
```

```
[(array([[ 0.0040953 ,  0.04639799],
       [ 0.05674624,  0.64290998],
       [-0.00898892, -0.10184046]]), array([ 0.05622079,  0.77901893,
-0.12340087])), (array([[0.00175181, 0.00429662, 0.00061569],
       [0.20018365, 0.49098549, 0.07035625],
       [0.06733028, 0.16513931, 0.0236638 ],
       [0.1903718 , 0.46692022, 0.06690779]]), array([0.00660111,
0.75432519, 0.25371165, 0.71735252]))]
```

```
[(array([[ 0.0040953 ,  0.04639799],
         [ 0.05674624,  0.64290998],
         [-0.00898892, -0.10184046]]),
  array([ 0.05622079,  0.77901893, -0.12340087])),
```

```
(array([[0.00175181, 0.00429662, 0.00061569],
        [0.20018365, 0.49098549, 0.07035625],
        [0.06733028, 0.16513931, 0.0236638 ],
        [0.1903718 , 0.46692022, 0.06690779]]),
 array([0.00660111, 0.75432519, 0.25371165, 0.71735252]))]
```

```python
# Exercise 6
def create_layers_batch(network_input_size, layer_output_sizes):
    layers = []

    i_size = network_input_size
    for layer_output_size in layer_output_sizes:
        W = np.random.randn(layer_output_size, i_size).T
        b = np.random.randn(layer_output_size)
        layers.append((W, b))

        i_size = layer_output_size
    return layers

def feed_forward_batch(inputs, layers, activation_funcs):
    a = inputs
    for (W, b), activation_func in zip(layers, activation_funcs):
        z = a @ W + b
        a = activation_func(z)
    return a

def cost_batch(layers, inputs, activation_funcs, target):
    predict = feed_forward_batch(inputs, layers, activation_funcs)
    return mse(predict, target)

def feed_forward_saver_batch(inputs, layers, activation_funcs):
    layer_inputs = []
    zs = []
    a = inputs
    for (W, b), activation_func in zip(layers, activation_funcs):
        layer_inputs.append(a)
        z = a @ W + b
        a = activation_func(z)

        zs.append(z)

    return layer_inputs, zs, a

def backpropagation_batch(inputs, layers, activation_funcs, target,
activation_ders, cost_der=mse_der):
    layer_inputs, zs, predict = feed_forward_saver_batch(inputs,
layers, activation_funcs)

    layer_grads = [None] * len(layers)

    for i in reversed(range(len(layers))):
```

```
        layer_input, z, activation_der = layer_inputs[i], zs[i],
activation_ders[i]

        if i == len(layers) - 1:
            # For last layer we use cost derivative as dC_da(L) can be
computed directly
            dC_da = cost_der(predict, target)
        else:
            # For other layers we build on previous z derivative, as
dC_da(i) = dC_dz(i+1) * dz(i+1)_da(i)
            (W, b) = layers[i + 1]
            dC_da = dC_dz @ W.T

        dC_dz = dC_da * activation_der(z)
        dC_dW = layer_input.T @ dC_dz / len(layers[-1][1])
        dC_db = np.mean(dC_dz, axis=0) / len(layers[-1][1]) *
len(layer_input) # deriv wrt b is 1

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads

number_of_datapoints = np.random.randint(2, 20)
network_input_size = np.random.randint(2, 20)
final_output_size = np.random.randint(2, 20)

inputs = np.random.rand(number_of_datapoints, network_input_size)
layer_output_sizes = [5, 2, final_output_size]
activation_funcs = [sigmoid, ReLU, sigmoid]
activation_ders = [sigmoid_der, ReLU_der, sigmoid_der]

layers = create_layers_batch(network_input_size, layer_output_sizes)

target = np.random.rand(number_of_datapoints, final_output_size)

layer_grads = backpropagation_batch(inputs, layers, activation_funcs,
target, activation_ders)

print("Number of datapoints:", number_of_datapoints)
print("Network input size:", network_input_size)
print("Final output size:", final_output_size)

print("Our gradients:")
for i in range(len(layer_grads)):
    print(i, layer_grads[i][1])

print("Autograd:")
cost_grad = grad(cost_batch, 0)
w_autograd = cost_grad(layers, inputs, activation_funcs, target)
for i in range(len(w_autograd)):
    print(i, w_autograd[i][1])
```

```
Number of datapoints: 17
Network input size: 19
Final output size: 7
Our gradients:
0 [ 1.68184154e-03 -2.09762290e-03  6.76325518e-03  4.85893527e-05
 -1.92748982e-03]
1 [0.         0.04003649]
2 [-0.00455653  0.00685776  0.01933652  0.00962605  0.01221039 -
0.00951721
 -0.00247036]
Autograd:
0 [ 1.68184154e-03 -2.09762290e-03  6.76325518e-03  4.85893527e-05
 -1.92748982e-03]
1 [0.         0.04003649]
2 [-0.00455653  0.00685776  0.01933652  0.00962605  0.01221039 -
0.00951721
 -0.00247036]

# Exercise 7a)
...

# Exercise 7b)
...
```

We were not able to complete exercise 7 in time, but will hopefully have time to update the submission during the weekend. If not, you will see the training implementation in the final project.