In [1]:
```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.diag(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)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 3
      1 import autograd.numpy as np  # We need to use this numpy wrapper to make automatic differentiation work later
      2 from autograd import grad, elementwise_grad
----> 3 from sklearn import datasets
      4 import matplotlib.pyplot as plt
      5 from sklearn.metrics import accuracy_score

ModuleNotFoundError: No module named 'sklearn'
```

Exercise 2a)

The shape of weights and biases will be...

In [494…
```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)
```

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

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

In [495…

```python
# 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)
```

```
[[0.00071956 0.04156912]
 [0.00119779 0.06919653]
 [0.0005225  0.03018487]] [0.04217315 0.07020202 0.03062348]
```

Exercise 3a)

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

In [496…

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

predict = a

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

print(mse_der(predict, target))

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

```
[0.23130578 0.49200586 0.13473569]
[0.23130578 0.49200586 0.13473569]
```

In [497…

```python
# Exercise 3c)
def sigmoid_der(z):
    return np.diag(np.exp(-z) / (1 + np.exp(-z))**2)

print(sigmoid_der(z))

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

```
[[0.1823264  0.         0.        ]
 [0.         0.14268532 0.        ]
 [0.         0.         0.22728557]]
[0.1823264  0.14268532 0.22728557]
```

In [498…
```python
# 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, 3)
```

In [499…
```python
# Exercise 3e)
dz_dW = np.tensordot(np.eye(len(target)), x, axes=0)
dz_db = np.ones(len(b))
```

In [500…
```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)
```

```
[[0.00071956 0.04156912]
 [0.00119779 0.06919653]
 [0.0005225  0.03018487]] [0.04217315 0.07020202 0.03062348]
```

In [501…
```python
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g, b_g)
```

```
[[0.00071956 0.04156912]
 [0.00119779 0.06919653]
 [0.0005225  0.03018487]] [0.04217315 0.07020202 0.03062348]
```

In [502…
```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)
```

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

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

In [503…

```python
# 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.

In [504…

```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

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

```
[[0.00186133 0.00228897]
 [0.00066551 0.00081841]
 [0.00173739 0.00213655]] [0.00319644 0.00114287 0.00298359]
[[0.01307715 0.01134609 0.00998665]
 [0.00605096 0.00524998 0.00462095]
 [0.00256883 0.00222878 0.00196174]
 [0.00504424 0.00437652 0.00385215]] [0.0160403  0.00742205 0.00315089 0.00618721]
```

In [505…

```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)
```

```python
    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)
```

Out[505…

```
[(array([[0.00186133, 0.00228897],
         [0.00066551, 0.00081841],
         [0.00173739, 0.00213655]]),
  array([0.00319644, 0.00114287, 0.00298359])),
 (array([[0.01307715, 0.01134609, 0.00998665],
         [0.00605096, 0.00524998, 0.00462095],
         [0.00256883, 0.00222878, 0.00196174],
         [0.00504424, 0.00437652, 0.00385215]]),
  array([0.0160403 , 0.00742205, 0.00315089, 0.00618721]))]
```

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.

In [506…

```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
```

```python
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
```

In [507…

```python
# 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
```

In [508…
```
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([[ 0.09724526,  0.15364005],
       [ 0.0366406 ,  0.05788934],
       [-0.15925263, -0.25160695]]), array([ 0.16196256,  0.06102514, -0.26523622])), (array([[0.        , 0.        , 0.        ],
       [0.06314564, 0.05086566, 0.05101922],
       [0.41967321, 0.33805906, 0.33907962],
       [0.02285746, 0.01841236, 0.01846794]]), array([0.        , 0.084113  , 0.55902472, 0.03044723]))]
```

Out[508…
```
[(array([[ 0.09724526,  0.15364005],
         [ 0.0366406 ,  0.05788934],
         [-0.15925263, -0.25160695]]),
  array([ 0.16196256,  0.06102514, -0.26523622])),
 (array([[0.        , 0.        , 0.        ],
         [0.06314564, 0.05086566, 0.05101922],
         [0.41967321, 0.33805906, 0.33907962],
         [0.02285746, 0.01841236, 0.01846794]]),
  array([0.        , 0.084113  , 0.55902472, 0.03044723]))]
```

In [509…
```
# Exercise 6
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 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(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 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(layers, input, activation_funcs, target):
    predict = feed_forward(input, layers, activation_funcs)
    return mse(predict, target)

def cost_batch(layers, inputs, activation_funcs, target):
    predict = feed_forward_batch(inputs, layers, activation_funcs)
    return np.sum(-target * np.log(predict)) # NOT THE CORRECT COST FUNCTION

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
```

In [510…

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

        print(dC_da)
        print(activation_der(z))
        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
```

In [511…

```
inputs = np.random.rand(10, 2)
network_input_size = 2
layer_output_sizes = [3, 4]
activation_funcs = [sigmoid, ReLU]
activation_ders = [sigmoid_der, ReLU_der]

layers = create_layers_batch(network_input_size, layer_output_sizes)

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

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

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

```
[[-0.02547315 -0.02836122 -0.02359569 -0.02044567 -0.02749925 -0.01731853
  -0.03517198 -0.03761136 -0.03630514 -0.03540921]
 [-0.04641806 -0.04641806 -0.04641806 -0.04641806 -0.04641806 -0.04641806
  -0.04641806 -0.04641806 -0.04641806 -0.04641806]
 [-0.10000785 -0.10000785 -0.10000785 -0.10000785 -0.10000785 -0.10000785
  -0.10000785 -0.10000785 -0.10000785 -0.10000785]
 [ 0.15618219  0.18558829  0.17317283  0.18189949  0.05345909  0.14726545
   0.04293775  0.0702213   0.17608983  0.15369665]]
[1 0 0 1]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[511], line 12
      9 x = np.random.rand(network_input_size)
     10 target = np.random.rand(4)
---> 12 layer_grads = backpropagation_batch(inputs, layers, activation_funcs, target, activation_ders)
     13 print(layer_grads)
     15 cost_grad = grad(cost, 0)

Cell In[510], line 18, in backpropagation_batch(input, layers, activation_funcs, target, activation_ders, cost_der)
     15 else:
     16     # For other layers we build on previous z derivative, as dC_da(i) = dC_dz(i+1) * dz(i+1)_da(i)
     17     (W, b) = layers[i + 1]
---> 18     dC_da = dC_dz @ W
     20 print(dC_da)
     21 print(activation_der(z))

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3
is different from 10)
```