Exercises week 42

October 13-17, 2025

Date: **Deadline is Friday October 17 at midnight**

Overarching aims of the exercises this week

The aim of the exercises this week is to train the neural network you implemented last week.

To train neural networks, we use gradient descent, since there is no analytical expression for the optimal parameters. This means you will need to compute the gradient of the cost function wrt. the network parameters. And then you will need to implement some gradient method.

You will begin by computing gradients for a network with one layer, then two layers, then any number of layers. Keeping track of the shapes and doing things step by step will be very important this week.

We recommend that you do the exercises this week by editing and running this notebook file, as it includes some checks along the way that you have implemented the neural network correctly, and running small parts of the code at a time will be important for understanding the methods. If you have trouble running a notebook, you can run this notebook in google colab instead(https://colab.research.google.com/drive/1FfvbN0XlhV-lATRPyGRTtTBnJr3zNuHL#offline=true&sandboxMode=true), though we recommend that you set up VSCode and your python environment to run code like this locally.

First, some setup code that you will need.

In [76]:
```
import autograd.numpy as np  # We need to use this numpy wrapper to make automatic
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 1 - Understand the feed forward pass

a) Complete last weeks' exercises if you haven't already (recommended).

Exercise 2 - Gradient with one layer using autograd

For the first few exercises, we will not use batched inputs. Only a single input vector is passed through the layer at a time.

In this exercise you will compute the gradient of a single layer. You only need to change the code in the cells right below an exercise, the rest works out of the box. Feel free to make changes and see how stuff works though!

a) If the weights and bias of a layer has shapes (10, 4) and (10), what will the shapes of the gradients of the cost function wrt. these weights and this bias be?

b) Complete the feed_forward_one_layer function. It should use the sigmoid activation function. Also define the weigth and bias with the correct shapes.

In [77]:
```
np.random.seed(314)
import autograd
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(3,2)
b = np.random.randn(3)
prediction = feed_forward_one_layer(W,b,x)
```

```
def prints():
    print(f"x = {x}")
    print("\n")
    print(f"target = {target}")
    print("\n")

    print(f"W = {W}")
    print("\n")

    print(f"b = {b}")
    print("\n")

    print(f"prediction = {prediction}")
```

c) Compute the gradient of the cost function wrt. the weigth and bias by running the cell below. You will not need to change anything, just make sure it runs by defining things correctly in the cell above. This code uses the autograd package which uses backprogagation to compute the gradient!

In [78]:
```
autograd_one_layer = autograd.grad(cost_one_layer, [0, 1])
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g)
print(b_g)
```

```
[[ 0.0487215   0.03127437]
 [-0.04837692 -0.03105318]
 [-0.01064921 -0.00683573]]
[ 0.05313874 -0.05276291 -0.01161469]
```

Exercise 3 - Gradient with one layer writing backpropagation by hand

Before you use the gradient you found using autograd, you will have to find the gradient "manually", to better understand how the backpropagation computation works. To do backpropagation "manually", you will need to write out expressions for many derivatives along the computation.

We want to find the gradient of the cost function wrt. the weight and bias. This is quite hard to do directly, so we instead use the chain rule to combine multiple derivatives which are easier to compute.

$$\frac{dC}{dW} = \frac{dC}{da}\frac{da}{dz}\frac{dz}{dW}$$

$$\frac{dC}{db} = \frac{dC}{da}\frac{da}{dz}\frac{dz}{db}$$

a) Which intermediary results can be reused between the two expressions?

We see that

$$\frac{dC}{da}\frac{da}{dz}$$

occur in both expression.

b) What is the derivative of the cost wrt. the final activation? You can use the autograd calculation to make sure you get the correct result. Remember that we compute the mean in mse.

The MSE cost function is:

$$C = \frac{1}{n}\sum_{i=1}^{n}(a_i - y_i)^2$$

Taking the derivative with respect to the activation a:

$$\frac{dC}{da} = \frac{2(a-y)}{n}$$

In [79]:
```
z = W @ x + b
a = sigmoid(z)

predict = a


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



print(mse_der(predict, target))

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

```
[ 0.22071046 -0.21208637 -0.07439771]
[ 0.22071046 -0.21208637 -0.07439771]
```

c) What is the expression for the derivative of the sigmoid activation function? You can use the autograd calculation to make sure you get the correct result.

The sigmoid function is:

$$a = \sigma(z) = \frac{1}{1+e^{-z}}$$

The derivative of the sigmoid with respect to z is:

$$\frac{da}{dz} = \sigma(z)(1-\sigma(z)) = a(1-a)$$

In [80]:
```
def sigmoid_der(z):
    a = sigmoid(z)
    return a * (1 - a)


print(sigmoid_der(z))

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

```
[0.24076221 0.2487803  0.15611628]
[0.24076221 0.2487803  0.15611628]
```

d) Using the two derivatives you just computed, compute this intermetidary gradient you will use later:

$$\frac{dC}{dz} = \frac{dC}{da}\frac{da}{dz}$$

So we have

$$\frac{da}{dz} = \sigma(z)(1-\sigma(z)) = a(1-a)$$

and

$$\frac{dC}{da} = \frac{2(a-y)}{n}$$

Multiplying these together:

$$\frac{dC}{dz} = \frac{dC}{da}\cdot\frac{da}{dz} = \frac{2(a-y)}{n}\cdot a(1-a) = \frac{2a(1-a)(a-y)}{n}$$

In [81]:
```
dC_da = mse_der(predict, target)
dC_dz = dC_da * sigmoid_der(z)
```

e) What is the derivative of the intermediary z wrt. the weight and bias? What should the shapes be? The one for the weights is a little tricky, it can be easier to play around in the next exercise first. You can also try computing it with autograd to get a hint.

So:

$$z = Wx + b$$

where W is 3×2, x is 2×1, b is 3×1, z is 3×1.

We have:

$$z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Expanding: $z_1 = W_{11}x_1 + W_{12}x_2 + b_1$, $z_2 = W_{21}x_1 + W_{22}x_2 + b_2$, $z_3 = W_{31}x_1 + W_{32}x_2 + b_3$

Weights W

Taking the derivative of z_i with respect to each element of W

$$\frac{dz_1}{dW_{11}} = x_1, \quad \frac{dz_1}{dW_{12}} = x_2$$

$$\frac{dz_2}{dW_{21}} = x_1, \quad \frac{dz_2}{dW_{22}} = x_2$$

$$\frac{dz_3}{dW_{31}} = x_1, \quad \frac{dz_3}{dW_{32}} = x_2$$

This gives us the Jacobian:

$$\frac{dz}{dW} = x^\top = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$$

And we can relate this to the derivative over the cost by the chain rule, i.e

$$\frac{dC}{dW} = \frac{dC}{dz}\frac{dz}{dW}$$

Bias b

Taking derivatives:

$$\frac{dz_1}{db_1} = 1, \quad \frac{dz_1}{db_2} = 0, \quad \frac{dz_1}{db_3} = 0$$

$$\frac{dz_2}{db_1} = 0, \quad \frac{dz_2}{db_2} = 1, \quad \frac{dz_2}{db_3} = 0$$

$$\frac{dz_3}{db_1} = 0, \quad \frac{dz_3}{db_2} = 0, \quad \frac{dz_3}{db_3} = 1$$

This gives us:

$$\frac{dz}{db} = I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

And we can relate this to the derivative over the cost by the chain rule, i.e

$$\frac{dC}{db} = \frac{dC}{dz}\frac{dz}{db} = \frac{dC}{dz}I_3$$

and we know that for any vector v, $I \cdot v = v$, hence

$$\frac{dC}{db} = \frac{dC}{dz}$$

which is a 3×1 vector, matching the shape of b.

f) Now combine the expressions you have worked with so far to compute the gradients! Note that you always need to do a feed forward pass while saving the zs and as before you do backpropagation, as they are used in the derivative expressions

In [82]:
```
z = W @ x + b
a = sigmoid(z)
predict = a
```

In [83]:
```
dC_da = mse_der(predict, target)
dC_dz = dC_da * sigmoid_der(z)
dC_dW = np.outer(dC_dz, x)
dC_db = dC_dz

print(dC_dW, dC_db)
```

```
[[ 0.0487215   0.03127437]
 [-0.04837692 -0.03105318]
 [-0.01064921 -0.00683573]] [ 0.05313874 -0.05276291 -0.01161469]
```

You should get the same results as with autograd.

In [84]:
```
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g, b_g)
```

```
[[ 0.0487215   0.03127437]
 [-0.04837692 -0.03105318]
 [-0.01064921 -0.00683573]] [ 0.05313874 -0.05276291 -0.01161469]
```

Exercise 4 - Gradient with two layers writing backpropagation by hand

Now that you have implemented backpropagation for one layer, you have found most of the expressions you will need for more layers. Let's move up to two layers.

In [85]:
```
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)]
```

In [86]:
```
z1 = W1 @ x + b1
a1 = sigmoid(z1)
z2 = W2 @ a1 + b2
a2 = sigmoid(z2)
```

We begin by computing the gradients of the last layer, as the gradients must be propagated backwards from the end.

a) Compute the gradients of the last layer, just like you did the single layer in the previous exercise.

In [87]:
```
dC_da2 = mse_der(a2, target)
dC_dz2 = dC_da2 * sigmoid_der(z2)
dC_dW2 = np.outer(dC_dz2, a1)
dC_db2 = dC_dz2
```

To find the derivative of the cost wrt. the activation of the first layer, we need a new expression, the one furthest to the right in the following.

$$\frac{dC}{da_1} = \frac{dC}{dz_2}\frac{dz_2}{da_1}$$

b) What is the derivative of the second layer intermetiate wrt. the first layer activation? (First recall how you compute z_2)

$$\frac{dz_2}{da_1}$$

Since

$$z_2 = W_2 a_1 + b_2$$

we take the derivative respect to a_1 over element $z_{2,i} = \sum_j W_{2,ij} a_{1,j} + b_{2,i}$ giving

$$\frac{dz_{2,i}}{da_{1,j}} = W_{2,ij}$$

Or matrix:

$$\frac{dz_2}{da_1} = W_2$$

c) Use this expression, together with expressions which are equivelent to ones for the last layer to compute all the derivatives of the first layer.

$$\frac{dC}{dW_1} = \frac{dC}{da_1}\frac{da_1}{dz_1}\frac{dz_1}{dW_1}$$

$$\frac{dC}{db_1} = \frac{dC}{da_1}\frac{da_1}{dz_1}\frac{dz_1}{db_1}$$

We already know from part (b) that:

$$\frac{dC}{da_1} = \frac{dC}{dz_2}\frac{dz_2}{da_1} = \frac{dC}{dz_2}W_2 = W_2^T\frac{dC}{dz_2}$$

From the activation function (same as layer 2):

$$\frac{da_1}{dz_1} = \sigma'(z_1) = a_1(1-a_1)$$

and from the linear transformation $z_1 = W_1x + b_1$ same as before:

$$\frac{dz_1}{dW_1} = x^T$$

$$\frac{dz_1}{db_1} = I$$

$$\frac{dC}{dz_1} = \frac{dC}{da_1}\frac{da_1}{dz_1} = \left(W_2^T\frac{dC}{dz_2}\right)\sigma'(z_1)$$

where the multiplication is element-wise since as the vectors have the same dimension.

Hence

$$\frac{dC}{dW_1} = \frac{dC}{dz_1}x^T$$

$$\frac{dC}{db_1} = \frac{dC}{dz_1}$$

In [88]:
```
dC_da1 = W2.T @ dC_dz2
dC_dz1 = dC_da1 * sigmoid_der(z1)
dC_dW1 = np.outer(dC_dz1, x)
dC_db1 = dC_dz1
```

In [89]:
```
print(dC_dW1, dC_db1)
print(dC_dW2, dC_db2)
```

```
[[0.00113143 0.00028436]
 [0.00150824 0.00037906]
 [0.00212856 0.00053496]] [0.00406726 0.00542184 0.00765174]
[[0.01028658 0.00774165 0.00780708]
 [0.00811599 0.00610807 0.00615969]
 [0.01307522 0.00984038 0.00992354]
 [0.00810123 0.00609697 0.00614849]] [0.0138624  0.01093727 0.01762043 0.01091738]
```

d) Make sure you got the same gradient as the following code which uses autograd to do backpropagation.

In [90]:
```
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
```

In [91]:
```
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[91]:
```
[(array([[0.00113143, 0.00028436],
         [0.00150824, 0.00037906],
         [0.00212856, 0.00053496]]),
  array([0.00406726, 0.00542184, 0.00765174])),
 (array([[0.01028658, 0.00774165, 0.00780708],
         [0.00811599, 0.00610807, 0.00615969],
         [0.01307522, 0.00984038, 0.00992354],
         [0.00810123, 0.00609697, 0.00614849]]),
  array([0.0138624 , 0.01093727, 0.01762043, 0.01091738]))]
```

e) How would you use the gradient from this layer to compute the gradient of an even earlier layer? Would the expressions be any different?

Exercise 5 - Gradient with any number of layers writing backpropagation by hand

Well done on getting this far! Now it's time to compute the gradient with any number of layers.

First, some code from the general neural network code from last week. Note that we are still sending in one input vector at a time. We will change it to use batched inputs later.

In [92]:
```
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)
```

You might have already have noticed a very important detail in backpropagation: You need the values from the forward pass to compute all the gradients! The feed forward method above is great for efficiency and for using autograd, as it only cares about computing the final output, but now we need to also save the results along the way.

Here is a function which does that for you.

In [93]:
```
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
```

a) Now, complete the backpropagation function so that it returns the gradient of the cost function wrt. all the weigths and biases. Use the autograd calculation below to make sure you get the correct answer.

In [94]:
```
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 dir
            dC_da = cost_der(predict, target)
        else:
            # For other layers we build on previous z derivative, as dC_da(i) = dC_
            (W, b) = layers[i + 1]
```

```
            dC_da = W.T @ dC_dz

        dC_dz = dC_da * activation_der(z)
        dC_dW = np.outer(dC_dz, layer_input)
        dC_db = dC_dz


        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads
```

In [95]:
```
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)
```

In [96]:
```
layer_grads = backpropagation(x, layers, activation_funcs, target, activation_ders)
print(layer_grads)
```

```
[(array([[-0.03411362, -0.10878774],
       [ 0.0097726 ,  0.03116463],
       [-0.00907399, -0.02893678]]), array([-0.11125824,  0.03187236, -0.0295939
2])), (array([[-0.04954165, -0.19894415, -0.26398387],
       [-0.        , -0.        , -0.        ],
       [ 0.00765355,  0.03073431,  0.04078211],
       [ 0.03266426,  0.13116972,  0.17405231]]), array([-0.42867566, -0.        ,
0.06622488,  0.28263845]))]
```

In [97]:
```
cost_grad = grad(cost, 0)
cost_grad(layers, x, [sigmoid, ReLU], target)
```

Out[97]:
```
[(array([[-0.03411362, -0.10878774],
         [ 0.0097726 ,  0.03116463],
         [-0.00907399, -0.02893678]]),
  array([-0.11125824,  0.03187236, -0.02959392])),
 (array([[-0.04954165, -0.19894415, -0.26398387],
         [ 0.        ,  0.        ,  0.        ],
         [ 0.00765355,  0.03073431,  0.04078211],
         [ 0.03266426,  0.13116972,  0.17405231]]),
  array([-0.42867566,  0.        ,  0.06622488,  0.28263845]))]
```

Exercise 6 - Batched inputs

Make new versions of all the functions in exercise 5 which now take batched inputs instead. See last weeks exercise 5 for details on how to batch inputs to neural networks. You will also need to update the backpropogation function.

In [98]:
```
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(i_size, layer_output_size)
        b = np.random.randn(layer_output_size)
        layers.append((W, b))
        i_size = layer_output_size
    return layers
```

In [99]:
```
def feed_forward_batch(inputs, layers, activation_funcs):
    a = inputs
    for (W, b), activation_func in zip(layers, activation_funcs):
        # (n_samples, n_features) @ (n_features, n_outputs)
        z = a @ W + b
        a = activation_func(z)
    return a
```

In [100…
```
def cost_batch(layers, inputs, activation_funcs, targets):
    predictions = feed_forward_batch(inputs, layers, activation_funcs)
    return mse(predictions, targets)
```

In [101…
```
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 [102…
```
def backpropagation_batch(
    inputs, layers, activation_funcs, targets, activation_ders, cost_der=mse_der
):
    layer_inputs, zs, predictions = feed_forward_saver_batch(inputs, layers, activa
    layer_grads = [() for _ in layers]
    n_samples = inputs.shape[0]

    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:
            dC_da = cost_der(predictions, targets)
        else:
            W, _ = layers[i + 1]
            dC_da = dC_dz @ W.T

        dC_dz = dC_da * activation_der(z)
        dC_dW = layer_input.T @ dC_dz / n_samples
        dC_db = np.mean(dC_dz, axis=0)
        layer_grads[i] = (dC_dW, dC_db)
```

```
    return layer_grads
```

In [103…

```
import autograd.numpy as np
from autograd import grad

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)

n_samples = 10
inputs = np.random.rand(n_samples, network_input_size)
targets = np.random.rand(n_samples, 4)

# manual gradients
manual_grads = backpropagation_batch(inputs, layers, activation_funcs, targets, act

# autograd gradients
cost_grad = grad(cost_batch, 0)
auto_grads = cost_grad(layers, inputs, activation_funcs, targets)

print("manual gradients:")
print(manual_grads)

print("\nautograd gradients:")
print(auto_grads)
```

```
manual gradients:
[(array([[ 0.00705068,  0.01351109, -0.00382829],
       [ 0.00723994,  0.01273161, -0.00414871]]), array([ 0.0099883 ,  0.02028504, -
0.0053293 ])), (array([[0.        , 0.0142052 , 0.        , 0.00380141],
       [0.        , 0.02456479, 0.        , 0.00773538],
       [0.        , 0.01412752, 0.        , 0.00443608]]), array([0.        , 0.0458
9897, 0.        , 0.01482161]))]

autograd gradients:
[(array([[ 0.07050681,  0.13511092, -0.03828287],
       [ 0.07239938,  0.12731607, -0.0414871 ]]), array([ 0.09988302,  0.20285044, -
0.05329297])), (array([[0.        , 0.14205205, 0.        , 0.03801415],
       [0.        , 0.24564794, 0.        , 0.07735377],
       [0.        , 0.14127524, 0.        , 0.04436078]]), array([0.        , 0.4589
8975, 0.        , 0.1482161 ]))]
```

Exercise 7 - Training

a) Complete exercise 6 and 7 from last week, but use your own backpropogation implementation to compute the gradient.

- IMPORTANT: Do not implement the derivative terms for softmax and cross-entropy separately, it will be very hard!

- Instead, use the fact that the derivatives multiplied together simplify to **prediction - target** (see source1, source2)

b) Use stochastic gradient descent with momentum when you train your network.

In [104…
```
def softmax(z):
    e = np.exp(z - np.max(z, axis=1, keepdims=True))
    return e / np.sum(e, axis=1, keepdims=True)

def cross_entropy(pred, target):
    return -np.sum(target * np.log(pred + 1e-9)) / len(target)
def ReLU(z):
    return np.maximum(0, z)

def d_ReLU(z):
    return (z > 0).astype(float)

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

def d_sigmoid(z):
    s = sigmoid(z)
    return s * (1 - s)
```

In [105…
```
def backpropagation_batch(
    inputs, layers, activation_funcs, targets, activation_ders
):
    layer_inputs, zs, predict = feed_forward_saver_batch(inputs, layers, activation
    grads = [() for _ in layers]
    n = inputs.shape[0]

    # simplified softmax-cross-entropy derivative
    dC_dz = (predict - targets) / n

    for i in reversed(range(len(layers))):
        a_prev = layer_inputs[i]
        dW = a_prev.T @ dC_dz
        db = np.mean(dC_dz, axis=0)
        grads[i] = (dW, db)
        if i > 0:
            W, _ = layers[i]
            z_prev = zs[i - 1]
            f_der = activation_ders[i - 1]
            dC_dz = (dC_dz @ W.T) * f_der(z_prev)
    return grads
```

In [106…
```
def train_network_batch(
    inputs,
    layers,
    activation_funcs,
    activation_ders,
    targets,
    learning_rate=0.01,
    epochs=500
```

```
):
    for epoch in range(epochs):
        grads = backpropagation_batch(inputs, layers, activation_funcs, targets, ac
        for (W, b), (dW, db) in zip(layers, grads):
            W -= learning_rate * dW
            b -= learning_rate * db

        if epoch % 100 == 0 or epoch == epochs - 1:
            preds = feed_forward_batch(inputs, layers, activation_funcs)
            loss = cross_entropy(preds, targets)
            acc = np.mean(np.argmax(preds, 1) == np.argmax(targets, 1))
            print("Epoch", epoch, "Loss:", round(loss, 4), "Acc:", round(acc, 3))
```

In [107…

```
def train_network_sgd_momentum(
    inputs,
    layers,
    activation_funcs,
    activation_ders,
    targets,
    learning_rate=0.01,
    momentum=0.9,
    batch_size=16,
    epochs=500
):
    velocities = [(np.zeros_like(W), np.zeros_like(b)) for (W, b) in layers]
    n = len(inputs)

    for epoch in range(epochs):
        idx = np.random.permutation(n)
        for start in range(0, n, batch_size):
            batch = idx[start:start + batch_size]
            Xb, Yb = inputs[batch], targets[batch]

            grads = backpropagation_batch(Xb, layers, activation_funcs, Yb, activat

            for j, ((W, b), (dW, db)) in enumerate(zip(layers, grads)):
                vW, vb = velocities[j]
                vW = momentum * vW - learning_rate * dW
                vb = momentum * vb - learning_rate * db
                W += vW
                b += vb
                layers[j] = (W, b)
                velocities[j] = (vW, vb)

        if epoch % 100 == 0 or epoch == epochs - 1:
            preds = feed_forward_batch(inputs, layers, activation_funcs)
            loss = cross_entropy(preds, targets)
            acc = np.mean(np.argmax(preds, 1) == np.argmax(targets, 1))
            print("Epoch", epoch, "Loss:", round(loss, 4), "Acc:", round(acc, 3))
```

In [115…

```
from sklearn import datasets
np.random.seed(314)

iris = datasets.load_iris()
X = iris.data
```

```
y = iris.target
Y = np.zeros((len(y), 3))
Y[np.arange(len(y)), y] = 1

np.random.seed(0)
layers = create_layers_batch(4, [12, 10, 3])
activation_funcs = [ReLU, ReLU, softmax]
activation_ders  = [d_ReLU, d_ReLU, None]

print("batc training:")
train_network_batch(X, layers, activation_funcs, activation_ders, Y, learning_rate=

print("\nstochastic :")
layers = create_layers_batch(4, [12, 10, 3])
train_network_sgd_momentum(X, layers, activation_funcs, activation_ders, Y, learnin
```

```
batc training:
Epoch 0 Loss: 13.8155 Acc: 0.333
Epoch 100 Loss: 0.1942 Acc: 0.967
Epoch 200 Loss: 0.1331 Acc: 0.967
Epoch 300 Loss: 0.1084 Acc: 0.967
Epoch 400 Loss: 0.0954 Acc: 0.967
Epoch 499 Loss: 0.0875 Acc: 0.973

stochastic :
Epoch 0 Loss: 2.8019 Acc: 0.553
Epoch 100 Loss: 0.2493 Acc: 0.96
Epoch 200 Loss: 0.1984 Acc: 0.96
Epoch 300 Loss: 0.1664 Acc: 0.96
Epoch 400 Loss: 0.1482 Acc: 0.96
Epoch 500 Loss: 0.1333 Acc: 0.96
Epoch 600 Loss: 0.1246 Acc: 0.96
Epoch 700 Loss: 0.1178 Acc: 0.967
Epoch 800 Loss: 0.1105 Acc: 0.967
Epoch 900 Loss: 0.107 Acc: 0.967
Epoch 999 Loss: 0.1004 Acc: 0.973
```

Exercise 8 (Optional) - Object orientation

Passing in the layers, activations functions, activation derivatives and cost derivatives into the functions each time leads to code which is easy to understand in isoloation, but messier when used in a larger context with data splitting, data scaling, gradient methods and so forth. Creating an object which stores these values can lead to code which is much easier to use.

a) Write a neural network class. You are free to implement it how you see fit, though we strongly recommend to not save any input or output values as class attributes, nor let the neural network class handle gradient methods internally. Gradient methods should be handled outside, by performing general operations on the layer_grads list using functions or classes separate to the neural network.

We provide here a skeleton structure which should get you started.

In [109…

```
class NeuralNetwork:
    def __init__(
        self,
        network_input_size,
        layer_output_sizes,
        activation_funcs,
        activation_ders,
        cost_fun,
        cost_der,
    ):
        pass

    def predict(self, inputs):
        # Simple feed forward pass
        pass

    def cost(self, inputs, targets):
        pass

    def _feed_forward_saver(self, inputs):
        pass

    def compute_gradient(self, inputs, targets):
        pass

    def update_weights(self, layer_grads):
        pass

    # These last two methods are not needed in the project, but they can be nice to
    def autograd_compliant_predict(self, layers, inputs):
        pass

    def autograd_gradient(self, inputs, targets):
        pass
```