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/1FfvbN0XIhV-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 [1]:
```
import autograd.numpy as np  # We need to use this numpy wrapper to make aut
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?

The gradient of the cost function with respect to the weights will have the same shape as the weigths, (10, 4). The gradient of the cost function wrt. the bias will have the same shape as the bias, (10)

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 [2]:
```
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.rand(3, 2)
b = np.random.rand(3)
```

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 [3]:
```
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)
print(W_g.shape)
print(b_g.shape)
```

```
[[ 0.02107676  0.0296198 ]
 [-0.00442246 -0.00621502]
 [ 0.00839028  0.01179112]] [ 0.03733446 -0.00783376  0.01486219]
(3, 2)
(3,)
```

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?

The terms $\frac{\partial C}{\partial a}$ and $\frac{\partial a}{\partial z}$ can be reused between the two expressions.

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.

We want to compute $\frac{\partial C}{\partial a} = \frac{\partial MSE(target,a)}{\partial a} = \frac{\partial \frac{1}{n}(target-a)^T(target-a)}{\partial a}$

From exercises week 35, we have that:

$$\frac{\partial (\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})^T(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})}{\partial \boldsymbol{s}} = -2(\boldsymbol{x}-\boldsymbol{A}\boldsymbol{s})^T\boldsymbol{A},$$

So we see that

$$\frac{\partial C}{\partial a} = -\frac{2}{n}(target-a)^T$$

In [4]:
```
z = W @ x + b
a = sigmoid(z)

predict = a


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


print(mse_der(predict, target))

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

```
[ 0.1646198  -0.0384867   0.10173114]
[ 0.1646198  -0.0384867   0.10173114]
```

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 activation function is defined as $\sigma(z) = \frac{1}{1+e^{-z}}$. Computing the derivative gives:

$$\frac{d\sigma(z)}{dz} = \frac{d}{dz}\frac{1}{1+e^{-z}} = \frac{e^{-z}}{(1+e^{-z})^2},$$

where I used the division rule for differentiation.

In [5]:
```
def sigmoid_der(z):
    return np.exp(-z)/((1+np.exp(-z))**2)


print(sigmoid_der(z))

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

```
[0.22679202 0.20354462 0.1460928 ]
[0.22679202 0.20354462 0.1460928 ]
```

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

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

In [6]:
```
dC_da = mse_der(predict, target)
da_dz = sigmoid_der(z)
dC_dz = dC_da*da_dz
print(dC_dz)
```

```
[ 0.03733446 -0.00783376  0.01486219]
```

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.

We start by writing out z component-wise:

$$z_k = (\sum_l W_{kl} * x_l) + b_k$$

We start by computing the derivative with respect to the weights:

$$\frac{\partial(\sum_l W_{kl} * x_l) + b_k}{\partial W_{ij}} = \frac{\sum_l W_{kl} * x_l}{\partial W_{ij}} = \sum_l \delta_{ki}\delta_{lj}x_l$$

$$= \delta_{ki}x_j,$$

where δ is the Kronecker-delta-notation.

We then compute the derivative with respect to the bias:

$$\frac{\partial(\sum_l W_{kl}x_l) + b_k}{\partial b_k} = \frac{\partial b_k}{\partial b_k} = 1,$$

so the derivative of the whole vector becomes a vector of ones, with the same shape as the bias vector.

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 [7]:
```
dC_da = mse_der(predict, target)
da_dz = sigmoid_der(z)
dC_dz = dC_da*da_dz


dC_dW = np.outer(dC_dz, x)
dC_db = dC_dz

print(dC_dW, dC_db)
```

```
[[ 0.02107676  0.0296198 ]
 [-0.00442246 -0.00621502]
 [ 0.00839028  0.01179112]] [ 0.03733446 -0.00783376  0.01486219]
```

You should get the same results as with autograd.

In [8]:
```
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g, b_g)
```

```
[[ 0.02107676  0.0296198 ]
 [-0.00442246 -0.00621502]
 [ 0.00839028  0.01179112]] [ 0.03733446 -0.00783376  0.01486219]
```

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 [9]:
```
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 [10]:
```
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 [11]:
```
dC_da2 = mse_der(predict=a2, target=target)
dC_dz2 = dC_da2*sigmoid_der(z2)
dC_dW2 = np.outer(dC_dz2, x)
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)

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

We compute z_2 in the following way: $z_2 = W_2 a_1 + b_2$. Differentiating this with respect to a_1 yields:

$$\frac{\partial z_2}{\partial a_1} = W_2$$

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.

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

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

In [12]:
```
dC_da1 = dC_dz2 @ W2
dC_dz1 = dC_da1 * sigmoid_der(z1)
dC_dW1 = np.outer(dC_dz1, x)
dC_db1 = dC_dz1
```

In [13]:
```
print(dC_dW1, dC_db1)
print(dC_dW2, dC_db2)
```

```
[[0.00365048 0.00271951]
 [0.00647092 0.00482066]
 [0.00790634 0.00589001]] [0.00514591 0.00912176 0.01114521]
[[0.02340415 0.01743546]
 [0.02405354 0.01791924]
 [0.01360597 0.01013608]
 [0.00836983 0.0062353 ]] [0.03299176 0.03390719 0.01917971 0.01179857]
```

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

In [14]:
```
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 [15]:
```
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[15]:
```
[(array([[0.00365048, 0.00271951],
         [0.00647092, 0.00482066],
         [0.00790634, 0.00589001]]),
  array([0.00514591, 0.00912176, 0.01114521])),
 (array([[0.02399616, 0.02514424, 0.02450553],
         [0.02466198, 0.02584192, 0.02518548],
         [0.01395013, 0.01461757, 0.01424625],
         [0.00858155, 0.00899213, 0.00876371]]),
  array([0.03299176, 0.03390719, 0.01917971, 0.01179857]))]
```

We see that the results are identical.

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?

We would compute the derivatives of an earlier layer (layer 0) in the following way. The derivatives can be written as follows:

$$\frac{dC}{dW_0} = \frac{dC}{da_0}\frac{da_0}{dz_0}\frac{dz_0}{dW_0},$$

$$\frac{dC}{db_0} = \frac{dC}{da_0}\frac{da_0}{dz_0}\frac{dz_0}{db_0}.$$

We also have the following expression:

$$\frac{dC}{da_0} = \frac{dC}{dz_1}\frac{dz_1}{da_0}.$$

We have already computed $\frac{dC}{dz_1}$, and $\frac{dz_1}{da_0}$ can be computed as we did in task b).

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 [16]:
```
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 [17]:
```
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 [18]:
```
def backpropagation(
    input, layers, activation_funcs, target, activation_ders, cost_der=mse_d
):
    layer_inputs, zs, predict = feed_forward_saver(input, layers, activation

    layer_grads = [() for layer in layers]

    dC_dz_next = None

    # 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_

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

        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)

        dC_dz_next = dC_dz

    return layer_grads
```

In [19]:
```
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 [20]:
```
layer_grads = backpropagation(x, layers, activation_funcs, target, activati
print(layer_grads)
```

```
[(array([[0.05712174, 0.06146033],
       [0.15741936, 0.16937591],
       [0.1730432 , 0.18618643]]), array([0.15057028, 0.41495022, 0.4561339
3])), (array([[ 0.06944416,  0.07843913,  0.11358062],
       [ 0.50702648,  0.57270069,  0.82927615],
       [-0.03769933, -0.04258245, -0.06165981],
       [-0.        , -0.        , -0.        ]]), array([ 0.16540557,  1.20
766105, -0.08979415, -0.        ]))]
```

In [21]:
```
cost_grad = grad(cost, 0)
cost_grad(layers, x, [sigmoid, ReLU], target)
```

Out[21]:
```
[(array([[0.05712174, 0.06146033],
         [0.15741936, 0.16937591],
         [0.1730432 , 0.18618643]]),
  array([0.15057028, 0.41495022, 0.45613393])),
 (array([[ 0.06944416,  0.07843913,  0.11358062],
         [ 0.50702648,  0.57270069,  0.82927615],
         [-0.03769933, -0.04258245, -0.06165981],
         [ 0.        ,  0.        ,  0.        ]]),
  array([ 0.16540557,  1.20766105, -0.08979415,  0.        ]))]
```

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 [22]:
```python
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(1, 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, input, activation_funcs, target):
    predict = feed_forward_batch(input, layers, activation_funcs)
    return mse(predict, target)

def feed_forward_saver_batch(input, layers, activation_funcs):
    layer_inputs = []
    zs = []
    a = input
    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 [23]:
```python
def backpropagation_batch(
    inputs, layers, activation_funcs, target, activation_ders, cost_der=mse_
):
    layer_inputs, zs, predict = feed_forward_saver_batch(inputs, layers, act

    layer_grads = [() for layer in layers]

    dC_dz_next = None
```

```
    B = inputs.shape[0]

    # 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_

        if i == len(layers) - 1:
            # For last layer we use cost derivative as dC_da(L) can be compu
            dC_da = cost_der(predict, target)
        else:
            # For other layers we build on previous z derivative, as dC_da(i
            (W_next, b_next) = layers[i + 1]
            dC_da = dC_dz_next @ W_next.T

        dC_dz = dC_da*activation_der(z)
        # Grads (average over batch)
        dC_dW = (layer_input.T @ dC_dz)
        dC_db = np.sum(dC_dz, axis=0, keepdims=True)

        layer_grads[i] = (dC_dW, dC_db)

        dC_dz_next = dC_dz

    return layer_grads
```

Testing/ comparing with autograd

Code snippet from ChatGPT

In [24]:
```
# ---- Print manual vs autograd gradients for easy side-by-side comparison -
# Assumes your functions (ReLU, ReLU_der, mse, backpropagation_batch, etc.)

import autograd.numpy as np
from autograd import grad

np.set_printoptions(precision=5, suppress=True)

# ---- toy setup (adjust sizes if you like) ----
B, in_dim, h_dim, out_dim = 6, 4, 5, 3
rng = np.random.RandomState(42)
X = rng.randn(B, in_dim)
Y = rng.randn(B, out_dim)

# Params in your shape convention: W:(in,out), b:(1,out)
W1 = rng.randn(in_dim, h_dim); b1 = rng.randn(1, h_dim)
W2 = rng.randn(h_dim, out_dim); b2 = rng.randn(1, out_dim)
layers = [(W1, b1), (W2, b2)]

# Use ReLU for hidden, identity for output
activation_funcs = [ReLU, lambda x: x]
activation_ders  = [ReLU_der, lambda x: np.ones_like(x)]

# Ensure the cost derivative has shape (B, out_dim)
def mse_der_batch(pred, tgt):
    return (2.0 / pred.size) * (pred - tgt)

# ---- your gradients ----
grads_manual = backpropagation_batch(
    X,
    layers,
    activation_funcs,
```

```
    Y,
    activation_ders,
    cost_der=mse_der_batch,   # use your mse_der if it already returns (B, c
)

# ---- autograd gradients on the exact same forward/loss ----
def forward_loss_autograd(W1_, b1_, W2_, b2_, X_, Y_):
    z1 = X_ @ W1_ + b1_
    a1 = ReLU(z1)
    pred = a1 @ W2_ + b2_
    return mse(pred, Y_)

gW1 = grad(forward_loss_autograd, 0)
gb1 = grad(forward_loss_autograd, 1)
gW2 = grad(forward_loss_autograd, 2)
gb2 = grad(forward_loss_autograd, 3)

dW1_ag = gW1(W1, b1, W2, b2, X, Y)
db1_ag = gb1(W1, b1, W2, b2, X, Y)
dW2_ag = gW2(W1, b1, W2, b2, X, Y)
db2_ag = gb2(W1, b1, W2, b2, X, Y)

grads_auto = [(dW1_ag, db1_ag), (dW2_ag, db2_ag)]

# ---- print and compare ----
for i, ((dW_np, db_np), (dW_ag, db_ag)) in enumerate(zip(grads_manual, grads
    print(f"\n=== Layer {i} ===")
    print("dW (manual):")
    print(dW_np)
    print("dW (autograd):")
    print(dW_ag)
    print("dW max|diff|:", float(np.max(np.abs(dW_np - dW_ag))))

    print("\ndb (manual):")
    print(db_np)
    print("db (autograd):")
    print(db_ag)
    print("db max|diff|:", float(np.max(np.abs(db_np - db_ag))))
```

```
=== Layer 0 ===
dW (manual):
[[ 0.1215   0.37245  1.05848  0.10538 -0.04399]
 [-0.0377   0.37243 -5.43553 -0.11608 -0.23706]
 [ 0.10893 -2.51195  6.2835   0.92487 -0.18495]
 [ 0.16942 -1.22071  7.72157  0.75858 -0.18267]]
dW (autograd):
[[ 0.1215   0.37245  1.05848  0.10538 -0.04399]
 [-0.0377   0.37243 -5.43553 -0.11608 -0.23706]
 [ 0.10893 -2.51195  6.2835   0.92487 -0.18495]
 [ 0.16942 -1.22071  7.72157  0.75858 -0.18267]]
dW max|diff|: 0.0

db (manual):
[[-0.11996 -1.59063 10.19279  0.9371   0.00465]]
db (autograd):
[[-0.11996 -1.59063 10.19279  0.9371   0.00465]]
db max|diff|: 0.0

=== Layer 1 ===
dW (manual):
[[-0.03238  0.03047  0.04005]
 [ 0.01171 -0.01716  0.00462]
 [ 5.0515  -7.80837  2.37407]
 [ 2.91909 -4.35392  1.08239]
 [ 0.1214  -1.37636  0.67007]]
dW (autograd):
[[-0.03238  0.03047  0.04005]
 [ 0.01171 -0.01716  0.00462]
 [ 5.0515  -7.80837  2.37407]
 [ 2.91909 -4.35392  1.08239]
 [ 0.1214  -1.37636  0.67007]]
dW max|diff|: 0.0

db (manual):
[[ 1.80103 -3.10111  0.8156 ]]
db (autograd):
[[ 1.80103 -3.10111  0.8156 ]]
db max|diff|: 0.0
```

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.

a)

In [25]:
```
# Set seed to compare results:
np.random.seed(1234)
```

In [26]:

```
from sklearn import datasets
from sklearn.metrics import accuracy_score

# Cross-entropy
def cross_entropy(predict, target):
    eps = 1e-12
    p = np.clip(predict, eps, 1.0 - eps)
    return np.sum(-target * np.log(p))

# Softmax
def softmax(z):
    z = z - np.max(z, axis=1, keepdims=True)
    e = np.exp(z)
    return e / np.sum(e, axis=1, keepdims=True)

# Accuracy function
def accuracy(predictions, targets):
    one_hot_predictions = np.zeros(predictions.shape)

    for i, prediction in enumerate(predictions):
        one_hot_predictions[i, np.argmax(prediction)] = 1
    return accuracy_score(one_hot_predictions, targets)

# Derivative for softmax + cross-entropy, only used in the last layer
def softmax_ce_der(pred, target):
    # Returns dL/da in the last layer: (y_hat - y) / B
    B = pred.shape[0]
    return (pred - target) / B

# Set up data (same as last week)
iris = datasets.load_iris()
X = iris.data.astype(float)
y = iris.target

# one-hot targets
Y = np.zeros((len(y), 3))
Y[np.arange(len(y)), y] = 1

# Standardizing
X = (X - X.mean(axis=0, keepdims=True)) / (X.std(axis=0, keepdims=True) + 1e

# Create layers (same as last week)
layers = create_layers_batch(4, [16, 3])

# Use softmax as last activation (same as week 41)
activation_funcs = [ReLU, softmax]

# Laste derivative = identity, so dL/dz = dL/da
activation_ders  = [ReLU_der, lambda z: np.ones_like(z)]

# Training (without momentum)
lr = 0.05
epochs = 300
batch_size = 32
n = X.shape[0]

def predict_fn(X_):
    a = X_
    for (W, b), act in zip(layers, activation_funcs):
        a = act(a @ W + b)
    return a

for epoch in range(1, epochs + 1):
```

```
    idx = np.random.permutation(n)
    Xs, Ys = X[idx], Y[idx]

    for start in range(0, n, batch_size):
        Xb = Xs[start:start + batch_size]
        Yb = Ys[start:start + batch_size]

        # Using backprop implemented earlier
        grads = backpropagation_batch(
            Xb,
            layers,
            activation_funcs,
            Yb,
            activation_ders,
            cost_der=softmax_ce_der,
        )

        # Parameter update (SGD)
        new_layers = []
        for (W, b), (dW, db) in zip(layers, grads):
            W_new = W - lr * dW
            b_new = b - lr * db
            new_layers.append((W_new, b_new))
        layers = new_layers

    # Logging
    if epoch % 50 == 0 or epoch == 1:
        preds = predict_fn(X)
        loss = cross_entropy(preds, Y)
        acc  = accuracy(preds, Y)
        print(f"Epoch {epoch:3d}/{epochs}  loss={loss:.2f}  acc={acc:.3f}")

# End evaluation
preds = predict_fn(X)
final_loss = cross_entropy(preds, Y)
final_acc  = accuracy(preds, Y)
print("\nFinal (full Iris): loss =", round(float(final_loss), 3), " acc =",
```

```
Epoch   1/300  loss=213.85  acc=0.700
Epoch  50/300  loss=10.72  acc=0.987
Epoch 100/300  loss=7.78  acc=0.993
Epoch 150/300  loss=6.61  acc=0.993
Epoch 200/300  loss=6.08  acc=0.993
Epoch 250/300  loss=5.70  acc=0.993
Epoch 300/300  loss=5.43  acc=0.993

Final (full Iris): loss = 5.428  acc = 0.993
```

b) With momentum

In [27]:

```
from sklearn import datasets
from sklearn.metrics import accuracy_score

# Cross-entropy
def cross_entropy(predict, target):
    eps = 1e-12
    p = np.clip(predict, eps, 1.0 - eps)
    return np.sum(-target * np.log(p))

# Softmax
def softmax(z):
    z = z - np.max(z, axis=1, keepdims=True)
    e = np.exp(z)
```

```
    return e / np.sum(e, axis=1, keepdims=True)

# Accuracy function
def accuracy(predictions, targets):
    one_hot_predictions = np.zeros(predictions.shape)

    for i, prediction in enumerate(predictions):
        one_hot_predictions[i, np.argmax(prediction)] = 1
    return accuracy_score(one_hot_predictions, targets)

# Derivative for softmax + cross-entropy, only used in the last layer
def softmax_ce_der(pred, target):
    # Returns dL/da in the last layer: (y_hat - y) / B
    B = pred.shape[0]
    return (pred - target) / B

# Set up data (same as last week)
iris = datasets.load_iris()
X = iris.data.astype(float)
y = iris.target

# one-hot targets
Y = np.zeros((len(y), 3))
Y[np.arange(len(y)), y] = 1

# Standardizing
X = (X - X.mean(axis=0, keepdims=True)) / (X.std(axis=0, keepdims=True) + 1e

# Create layers (same as last week)
layers = create_layers_batch(4, [16, 3])

# Use softmax as last activation (same as week 41)
activation_funcs = [ReLU, softmax]

# Laste derivative = identity, so dL/dz = dL/da
activation_ders  = [ReLU_der, lambda z: np.ones_like(z)]

# Training (with momentum)
lr = 0.05
beta = 0.9 # Momentum
epochs = 300
batch_size = 32
n = X.shape[0]
velocities = [(np.zeros_like(W), np.zeros_like(b)) for (W, b) in layers]

def predict_fn(X_):
    a = X_
    for (W, b), act in zip(layers, activation_funcs):
        a = act(a @ W + b)
    return a

for epoch in range(1, epochs + 1):
    idx = np.random.permutation(n)
    Xs, Ys = X[idx], Y[idx]

    for start in range(0, n, batch_size):
        Xb = Xs[start:start + batch_size]
        Yb = Ys[start:start + batch_size]

        # Using backprop implemented earlier
        grads = backpropagation_batch(
            Xb,
            layers,
            activation_funcs,
```

```
            Yb,
            activation_ders,
            cost_der=softmax_ce_der,
        )

        # Parameter update (SGD with momentum)
        new_layers = []
        new_vels = []
        for (W, b), (dW, db), (vW, vB) in zip(layers, grads, velocities):
            vW_new = beta * vW - lr * dW
            vB_new = beta * vB - lr * db
            W_new = W + vW_new
            b_new = b + vB_new
            new_layers.append((W_new, b_new))
            new_vels.append((vW_new, vB_new))
        layers = new_layers
        velocities = new_vels

    # Logging
    if epoch % 50 == 0 or epoch == 1:
        preds = predict_fn(X)
        loss = cross_entropy(preds, Y)
        acc  = accuracy(preds, Y)
        print(f"Epoch {epoch:3d}/{epochs}  loss={loss:.2f}  acc={acc:.3f}")

# End evaluation
preds = predict_fn(X)
final_loss = cross_entropy(preds, Y)
final_acc  = accuracy(preds, Y)
print("\nFinal (full Iris): loss =", round(float(final_loss), 3), " acc =",
```

```
Epoch   1/300  loss=55.32  acc=0.867
Epoch  50/300  loss=5.83  acc=0.987
Epoch 100/300  loss=5.05  acc=0.993
Epoch 150/300  loss=4.10  acc=0.987
Epoch 200/300  loss=3.50  acc=0.987
Epoch 250/300  loss=3.03  acc=0.987
Epoch 300/300  loss=2.52  acc=0.993

Final (full Iris): loss = 2.516  acc = 0.993
```

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 [28]:

```python
from typing import List, Tuple, Callable, Optional

Array = np.ndarray
Layer = Tuple[Array, Array]
Grads = List[Tuple[Array, Array]]

class NeuralNetwork:
    def __init__(
        self,
        network_input_size: int,
        layer_output_sizes: List[int],
        activation_funcs: List[Callable[[Array], Array]],
        activation_ders: List[Callable[[Array], Array]],
        cost_fun: Callable,
        cost_der: Callable,
    ):
        """Setting up neural network with given layers, activation functions
        and cost function.

        Args:
            network_input_size (int):
                Number of input neurons
            layer_output_sizes (List[int]):
                List containing number of nodes in each layer
            activation_funcs (List[Callable[[Array], Array]]):
                List of activation functions (one for each layer)
            activation_ders (List[Callable[[Array], Array]]):
                List of derivatives of activation functions
            cost_fun (Callable):
                Cost function.
            cost_der (Callable):
                Derivative of cost function
        """
        # Checking that the parameters line up (shapes):
        assert len(layer_output_sizes) == len(activation_funcs) == len(activ
            "Number of layers, activation functions and derivatives of activ
        )

        self.activation_funcs = activation_funcs
        self.activation_ders = activation_ders
        self.cost_fun = cost_fun
        self.cost_der = cost_der

        self.layers = create_layers_batch(network_input_size, layer_output_s

    def predict(self, inputs):
        """Perform a forward pass through the neural network

        Args:
            inputs (Array):
                Input data of shape (B, in_dim), where B is the batch size

        Returns:
        np.ndarray:
            The network output after the final activation function. Typical
            represents probabilities if the last activation is a softmax lay
        """
        a = inputs
        for (W, b), act in zip(self.layers, self.activation_funcs):
            a = act(a @ W + b)

        return a
```

```
    def cost(self, inputs, targets):
        """ Compute the loss for a given batch using the network's configure
        loss function.

        Args:
            inputs (np.ndarray):
                Input data of shape (B, in_dim), where B is the batch size
            targets (np.ndarray):
                Target labels in one-hot (or appropriate) format matching th
                output shape of the network, typically (B, out_dim).

        Returns:
        float:
            The scalar loss value computed by `self.cost_fun`
        """
        preds = self.predict(inputs)
        return self.cost_fun(preds, targets)

    def _feed_forward_saver(self, inputs):
        """Perform a forward pass while storing intermediate values
        needed for backpropagation.

        Args:
            inputs (np.ndarray):
                Input data of shape (B, in_dim), where B is the batch size.

        Returns:
        tuple:
            (layer_inputs, zs, a)
            - layer_inputs: list of activations before each layer (a_l)
            - zs: list of pre-activation values (z_l)
            - a: final output after the last activation
        """
        layer_inputs = []
        zs = []
        a = inputs

        for (W, b), act in zip(self.layers, self.activation_funcs):
            layer_inputs.append(a)

            z = a @ W + b
            zs.append(z)

            a = act(z)

        return layer_inputs, zs, a

    def compute_gradient(self, inputs, targets):
        """Compute parameter gradients for a given batch using the configure
        cost derivative.

        This method is a thin wrapper around the existing `backpropagation_b
        implementation. It does not perform any optimization step; it only r
        gradients so that an external optimizer can produce weight updates.

        Args:
            inputs (np.ndarray):
                Input batch of shape (B, in_dim).
            targets (np.ndarray):
                Target batch of shape (B, out_dim), typically one-hot for cl

        Returns:
        list[tuple[np.ndarray, np.ndarray]]:
            A list of (dW, db) tuples, one per layer, matching `self.layers`
```

```
            Shapes:
              - dW: (in_dim_l, out_dim_l)
              - db: (1, out_dim_l)

        Notes:
        - Uses `self.cost_der` provided at construction time (e.g., softmax
        - For softmax + cross-entropy, ensure your last activation derivati
          (so you don't multiply by the softmax derivative again).
        """
        return backpropagation_batch(
            inputs=inputs,
            layers=self.layers,
            activation_funcs=self.activation_funcs,
            activation_ders=self.activation_ders,
            target=targets,
            cost_der=self.cost_der
            )

    def update_weights(self, layer_grads):
        """Apply parameter updates to the network layers.

        Args:
            layer_grads : list[tuple[np.ndarray, np.ndarray]]
                A list of (dW, db) tuples, one per layer, matching the orde
                IMPORTANT: These should be *updates* (deltas), not raw grad
                For example:
                - Plain SGD:           (dW, db) = (-lr * gradW, -lr * gradb
                - SGD with momentum:   (dW, db) = (vW, vB) where v = beta*v

        Notes:
        - This method does not implement any optimization logic (no lr, no m
          It simply adds the provided updates to the current weights and bia
        - Shapes must match each layer:
              W: (in_dim_l, out_dim_l),  b: (1, out_dim_l)
             dW: (in_dim_l, out_dim_l), db: (1, out_dim_l)
        """
        new_layers = []
        for (W, b), (dW, db) in zip(self.layers, layer_grads):
            new_W = W + dW
            new_b = b + db
            new_layers.append((new_W, new_b))

        self.layers = new_layers

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

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