# 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 automat
from autograd import grad, elementwise_grad
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

np.random.seed(42)

# 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 softmax(z):
    e_z = np.exp(z - np.max(z, axis=0))
    return e_z / np.sum(e_z, axis=1)[:, np.newaxis]


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 [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(4)
target = np.random.rand(10)

W = np.random.randn(10, 4)
b = np.random.randn(10)
```

**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)
```

```
[[ 0.00083613  0.00212239  0.00163411  0.00133645]
 [-0.00034954 -0.00088724 -0.00068312 -0.00055869]
 [ 0.00159815  0.00405667  0.0031234   0.00255446]
 [-0.00873114 -0.02216271 -0.01706398 -0.01395571]
 [-0.00029545 -0.00074996 -0.00057742 -0.00047224]
 [ 0.001371    0.00348007  0.00267945  0.00219138]
 [ 0.00018229  0.0004627   0.00035626  0.00029136]
 [-0.00868816 -0.0220536  -0.01697997 -0.013887  ]
 [-0.00722805 -0.01834733 -0.01412636 -0.01155319]
 [ 0.00763695  0.01938526  0.01492551  0.01220677]] [ 0.00223241 -0.00093324  0.0
0426697 -0.02331164 -0.00078884  0.00366048
  0.00048669 -0.02319687 -0.01929847  0.02039021]
```

# 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?

**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.

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

predict = a
```

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


print(mse_der(predict, target))

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

```
[ 0.01297222 -0.00974552  0.02725206 -0.12137639 -0.00324895  0.03059018
  0.00833571 -0.09280045 -0.13034595  0.10390521]
[ 0.01297222 -0.00974552  0.02725206 -0.12137639 -0.00324895  0.03059018
  0.00833571 -0.09280045 -0.13034595  0.10390521]
```

**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.

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))
```

```
[0.17209188 0.09576072 0.15657435 0.19206071 0.24279704 0.11966205
 0.05838637 0.2499651  0.14805574 0.19623856]
[0.17209188 0.09576072 0.15657435 0.19206071 0.24279704 0.11966205
 0.05838637 0.2499651  0.14805574 0.19623856]
```

**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 = 2 * (predict - target)
dC_dz = dC_da * (np.exp(-z) / (1 + np.exp(-z))**2)
```

**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.

**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 = 2 * (predict - target) / len(target)
dC_dz = dC_da * (predict * (1 - predict))
dC_dW = np.outer(dC_dz, x)
dC_db = dC_dz

print(dC_dW, dC_db)
```

```
[[ 0.00083613  0.00212239  0.00163411  0.00133645]
 [-0.00034954 -0.00088724 -0.00068312 -0.00055869]
 [ 0.00159815  0.00405667  0.0031234   0.00255446]
 [-0.00873114 -0.02216271 -0.01706398 -0.01395571]
 [-0.00029545 -0.00074996 -0.00057742 -0.00047224]
 [ 0.001371    0.00348007  0.00267945  0.00219138]
 [ 0.00018229  0.0004627   0.00035626  0.00029136]
 [-0.00868816 -0.0220536  -0.01697997 -0.013887  ]
 [-0.00722805 -0.01834733 -0.01412636 -0.01155319]
 [ 0.00763695  0.01938526  0.01492551  0.01220677]] [ 0.00223241 -0.00093324  0.0
0426697 -0.02331164 -0.00078884  0.00366048
  0.00048669 -0.02319687 -0.01929847  0.02039021]
```

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)
```

```
[[ 0.00083613  0.00212239  0.00163411  0.00133645]
 [-0.00034954 -0.00088724 -0.00068312 -0.00055869]
 [ 0.00159815  0.00405667  0.0031234   0.00255446]
 [-0.00873114 -0.02216271 -0.01706398 -0.01395571]
 [-0.00029545 -0.00074996 -0.00057742 -0.00047224]
 [ 0.001371    0.00348007  0.00267945  0.00219138]
 [ 0.00018229  0.0004627   0.00035626  0.00029136]
 [-0.00868816 -0.0220536  -0.01697997 -0.013887  ]
 [-0.00722805 -0.01834733 -0.01412636 -0.01155319]
 [ 0.00763695  0.01938526  0.01492551  0.01220677]] [ 0.00223241 -0.00093324  0.0
0426697 -0.02331164 -0.00078884  0.00366048
  0.00048669 -0.02319687 -0.01929847  0.02039021]
```

# 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 = 2 * (a2 - target)
dC_dz2 = dC_da2 * (a2 * (1 - a2))
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 $z_2$)

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

In [12]:
```
dz2_da1 = dC_dz2 * W2.T
```

**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 [13]:
```
dC_da1 = W2.T @ dC_dz2
dC_dz1 = dC_da1 * (a1 * (1 - a1))
dC_dW1 = np.outer(dC_dz1, x)
dC_db1 = dC_dz1
```

In [14]:
```
print(dC_dW1, dC_db1)
print(dC_dW2, dC_db2)
```

```
[[0.00612014 0.00117555]
 [0.01233333 0.00236897]
 [0.00581621 0.00111717]] [0.01849555 0.03727229 0.01757704]
[[0.10902333 0.09323959 0.10071493]
 [0.11922086 0.10196078 0.11013533]
 [0.01405453 0.0120198  0.01298347]
 [0.03959955 0.03386656 0.03658176]] [0.1455121  0.15912262 0.01875841 0.0528530
3]
```

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

In [15]:
```
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 [16]:
```
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[16]:
```
[(array([[0.00153004, 0.00029389],
         [0.00308333, 0.00059224],
         [0.00145405, 0.00027929]]),
  array([0.00462389, 0.00931807, 0.00439426])),
 (array([[0.02725583, 0.0233099 , 0.02517873],
         [0.02980521, 0.0254902 , 0.02753383],
         [0.00351363, 0.00300495, 0.00324587],
         [0.00989989, 0.00846664, 0.00914544]]),
  array([0.03637803, 0.03978065, 0.0046896 , 0.01321326]))]
```

**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 [17]:
```
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 [18]:
```
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 [19]:
```
def backpropagation(
    input, layers, activation_funcs, target, activation_ders, cost_der=mse_der
):
    layer_inputs, zs, predict = feed_forward_saver(input, layers, activation_fun

    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

        if i == len(layers) - 1:
            # For last layer we use cost derivative as dC_da(L) can be computed
            dC_da = cost_der(predict, target)
        else:
            # For other layers we build on previous z derivative, as dC_da(i) =
            (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 [20]:
```
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 [21]:
```
layer_grads = backpropagation(x, layers, activation_funcs, target, activation_de
print(layer_grads)
```

```
[(array([[-0.00233957, -0.0386618 ],
       [ 0.00702544,  0.11609682],
       [-0.00193342, -0.03195012]]), array([-0.06342531,  0.19045868, -0.0524146
7])), (array([[ 0.43361218,  0.50798478,  0.34213817],
       [-0.        , -0.        , -0.        ],
       [-0.        , -0.        , -0.        ],
       [-0.        , -0.        , -0.        ]]), array([ 0.71988719, -0.
, -0.        , -0.        ]))]
```

In [22]:
```
cost_grad = grad(cost, 0)
cost_grad(layers, x, [sigmoid, ReLU], target)
```

Out[22]:
```
[(array([[-0.00233957, -0.0386618 ],
         [ 0.00702544,  0.11609682],
         [-0.00193342, -0.03195012]]),
  array([-0.06342531,  0.19045868, -0.05241467])),
 (array([[0.43361218, 0.50798478, 0.34213817],
         [0.        , 0.        , 0.        ],
         [0.        , 0.        , 0.        ],
         [0.        , 0.        , 0.        ]]),
  array([0.71988719, 0.        , 0.        , 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 [23]:
```python
# Importing the batch layer creator from week 41
def create_layers_batch(network_input_size : int, layer_output_sizes : list):
    layers = []

    # Number of inputs in the current layer
    i_size = network_input_size

    # For each output layer size
    for layer_output_size in layer_output_sizes:

        # w has the shape of the current output layer size x the current input s
        W = np.random.randn(i_size, layer_output_size)

        # b has the shape of the current output layer size, 1
        b = np.random.randn(1, layer_output_size)

        # Append to the layer list
        layers.append((W, b))

        # Update i_size to the output size of the current layer
        i_size = layer_output_size
    return layers
```

In [24]:
```python
def feed_forward_batch_saver(inputs, layers : list, activation_funcs : list):
    layer_inputs = []
    zs = []

    # Set the current a to the input vector
    a = inputs

    # For each layer and activation function
    for (W, b), activation_func in zip(layers, activation_funcs):
        layer_inputs.append(a)

        # Calculate z for the current W and b, and the previous a
        z = a @ W + b

        # Calculate a using the given activation function
        a = activation_func(z)
```

```
        zs.append(z)

    return layer_inputs, zs, a

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

    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

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

        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 [25]:
```
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(1, network_input_size)
target = np.random.rand(1, 4)
```

In [26]:
```
layer_grads = backpropagation_batch(x, layers, activation_funcs, target, activat
print(layer_grads)
```

```
[(array([[-0.03896774, -0.0943856 ],
       [-0.23900756, -0.57891152],
       [ 0.31565587,  0.76456503]]), array([[-0.10076073, -0.61801323,  0.8162064
2]])), (array([[ 0.65947035,  0.77276051,  2.27933059],
       [-0.        , -0.        , -0.        ],
       [ 0.28332828,  0.33200114,  0.97926893],
       [-0.        , -0.        , -0.        ]]), array([[ 2.88283571, -0.
,  1.23855287, -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)

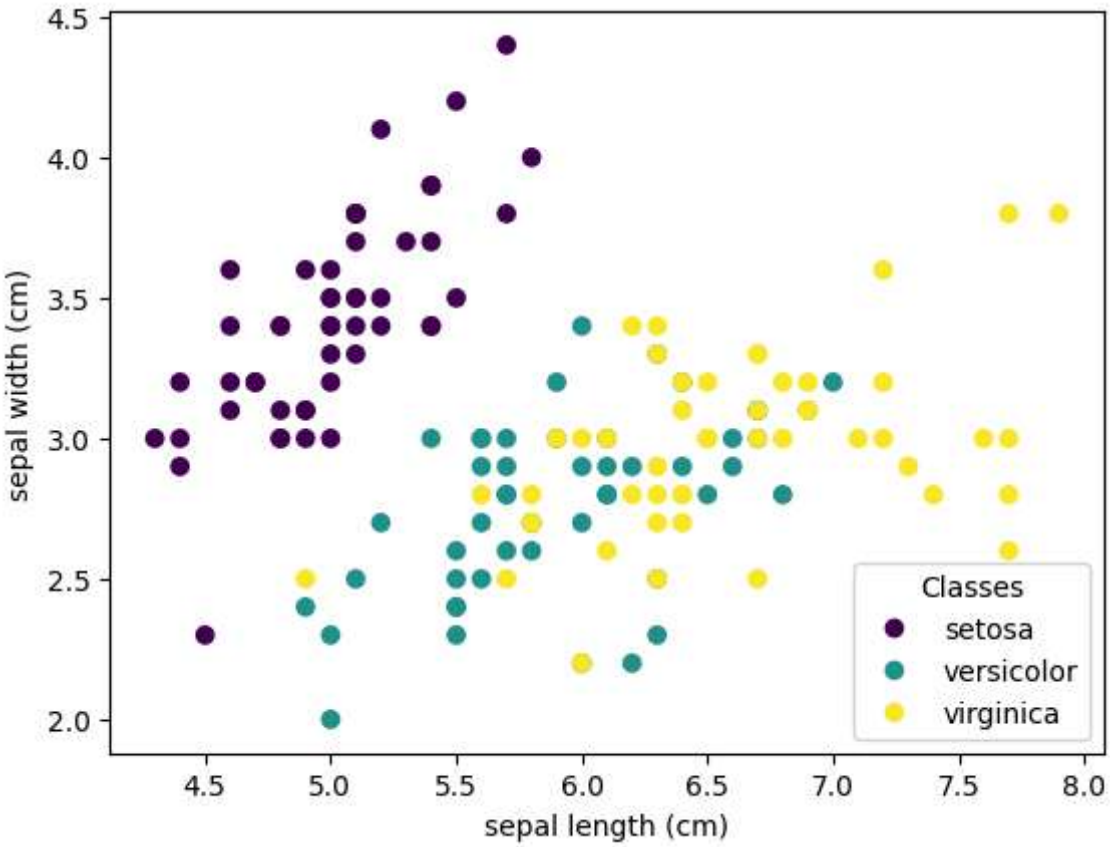
**b)** Use stochastic gradient descent with momentum when you train your network.

In [27]:
```
# Importing Iris data and functions from week 41
iris = datasets.load_iris()

_, ax = plt.subplots()
scatter = ax.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)
ax.set(xlabel=iris.feature_names[0], ylabel=iris.feature_names[1])
_ = ax.legend(
    scatter.legend_elements()[0], iris.target_names, loc="lower right", title="C
)

inputs = iris.data

# Since each prediction is a vector with a score for each of the three types of
# we need to make each target a vector with a 1 for the correct flower and a 0 f
targets = np.zeros((len(iris.data), 3))
for i, t in enumerate(iris.target):
    targets[i, t] = 1
```




In [28]:
```
inputs = iris.data
activation_funcs = [sigmoid, softmax]
network_input_size = 4
layer_output_sizes = [8, 3]
layers = create_layers_batch(network_input_size, layer_output_sizes)
```

In [29]:
```
def backpropagation_batch(
    input, layers, activation_funcs, target, activation_ders, cost_der=mse_der
):
    layer_inputs, zs, predict = feed_forward_batch_saver(input, layers, activati

    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

        if i == len(layers) - 1:
            # For last layer we use cost derivative as dC_da(L) can be computed
            dC_dz = predict - target # cross entropy softmax
        else:
            # For other layers we build on previous z derivative, as dC_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(input)
        dC_db = dC_dz

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads
```

In [30]:
```
layer_inputs, zs, predictions = feed_forward_batch_saver(inputs, layers, activat
```

In [31]:
```
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)

print(accuracy(predictions, targets))
```

```
0.6533333333333333
```

# 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 [32]:
```
class NeuralNetwork:
    def __init__(
        self,
        network_input_size,
        layer_output_sizes,
        activation_funcs,
        cost_func,
    ):
        self.network_input_size = network_input_size
        self.layer_output_sizes = layer_output_sizes
        self.activation_funcs = activation_funcs
        self.cost_func = cost_func
        self.layers = self.create_layers()
        self.gradient_func = self._create_gradient_func()
        self.velocities = None


    def create_layers(self):
        layers = []
        i_size = self.network_input_size

        for layer_output_size in self.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 predict(self, inputs):
        a = inputs

        for (W, b), activation_func in zip(self.layers, self.activation_funcs):
            z = a @ W + b
            a = activation_func(z)

        return a


    def _create_gradient_func(self):
        from autograd import grad

        def cost(inputs, layers, activation_funcs, targets):
            a = inputs
            for (W, b), activation_func in zip(layers, activation_funcs):
                z = a @ W + b
                a = activation_func(z)
            return self.cost_func(a, targets)

        gradient_func = grad(cost, 1)
        return gradient_func


    def compute_gradients(self, inputs, targets):
        layers_grad = self.gradient_func(inputs, self.layers, self.activation_fu
```

```
        return layers_grad


    def update_weights(self, layers_grad, eta=0.001):
        for j, ((W, b), (W_g, b_g)) in enumerate(zip(self.layers, layers_grad)):
            W -= W_g * eta
            b -= b_g * eta
            self.layers[j] = (W, b)


    ### Adapt GDM and SGD functions from extra work week 41 ###

    def update_weights_momentum(self, layers_grad, eta=0.001, alpha=0.9):
        # Initialize velocities to 0
        if self.velocities is None:
            self.velocities = []
            for W, b in self.layers:
                W_v = np.zeros_like(W)
                b_v = np.zeros_like(b)
                self.velocities.append((W_v, b_v))


        # For each weight and bias, and their corresponding gradient
        for indx, ((W, b), (W_g, b_g), (W_v, b_v)) in enumerate(zip(self.layers,

            # Compute the velocity of the weight and bias
            W_v = alpha * W_v - eta * W_g
            b_v = alpha * b_v - eta * b_g

            # Compute the new weight and bias values
            W_n = W + W_v
            b_n = b + b_v

            # Update weight and bias of the current layer
            self.layers[indx] = (W_n, b_n)
            self.velocities[indx] = (W_v, b_v)


    def update_weights_stochastic_momentum(self, inputs, targets, eta=0.001, alp
        # Initialize velocities only once (on first call)
        if self.velocities is None:
            self.velocities = []
            for W, b in self.layers:
                W_v = np.zeros_like(W)
                b_v = np.zeros_like(b)
                self.velocities.append((W_v, b_v))

        # Shuffle data and targets
        p = np.random.permutation(len(inputs))
        shuffled_inputs = inputs[p]
        shuffled_targets = targets[p]

        # Iterate through the data in 'batch_size' steps
        for j in range(0, len(inputs), batch_size):
            # Slices out a part of the data equal to the 'batch_size'
            batch_inputs = shuffled_inputs[j : j + batch_size]
            batch_targets = shuffled_targets[j : j + batch_size]

            # Find the gradients of each layer
            layers_grad = self.gradient_func(batch_inputs, self.layers, self.act
```

```
            # For each weight and bias, and their corresponding gradient
            for indx, ((W, b), (W_g, b_g), (W_v, b_v)) in enumerate(zip(self.lay
                # Compute the velocity of the weight and bias
                W_v = alpha * W_v - eta * W_g
                b_v = alpha * b_v - eta * b_g

                # Compute the new weight and bias values
                W_n = W + W_v
                b_n = b + b_v

                # Update weight and bias of the current layer
                self.layers[indx] = (W_n, b_n)
                self.velocities[indx] = (W_v, b_v)
```

In [33]:
```
def train_network(neural_network, inputs, targets, eta=0.001, epochs=100):
    for i in range(epochs):
        layers_grad = neural_network.compute_gradients(inputs, targets)
        neural_network.update_weights(layers_grad, eta)

    return neural_network.layers

def train_network_momentum(neural_network, inputs, targets, eta=0.001, alpha=0.9
    for i in range(epochs):
        layers_grad = neural_network.compute_gradients(inputs, targets)
        neural_network.update_weights_momentum(layers_grad, eta, alpha)

    return neural_network.layers

def train_network_stochastic_momentum(neural_network, inputs, targets, eta=0.001
    for i in range(epochs):
        layers_grad = neural_network.compute_gradients(inputs, targets)
        neural_network.update_weights_stochastic_momentum(layers_grad, eta, alph

    return neural_network.layers
```

In [34]:
```
network_input_size = 4
layer_output_sizes = [8, 3]
activation_funcs = [sigmoid, softmax]

def cross_entropy(predict, target):
    return np.sum(-target * np.log(predict))
```

In [35]:
```
NN = NeuralNetwork(network_input_size, layer_output_sizes, activation_funcs, cro
```

In [36]:
```
predictions = NN.predict(inputs)

print(accuracy(predictions, targets))
```

```
0.0
```

In [37]:
```
train_network_momentum(NN, inputs, targets, epochs=150)
```

Out[37]:
```
[(array([[-0.68432677, -1.36414979,  3.84101582,  1.53926573, -2.20229406,
           1.32050831,  1.51234898,  0.11419918],
         [-1.15602205, -1.72146001,  0.69948375,  0.20079463, -2.31772575,
          -0.04613368, -0.16913733, -2.65750604],
         [ 2.17566116, -1.71000004,  0.51762144, -0.80585824,  2.95451703,
           1.55693434,  0.25866512,  2.24125645],
         [-0.13297511,  1.88174666,  1.80525977,  1.38864371,  3.69228682,
           0.01458189,  0.45606437,  1.74673622]]),
  array([[-1.10973979, -0.63597001, -1.04284836, -0.75413384,  0.00394422,
          -0.33640482, -0.20897553,  0.15857356]])),
 (array([[-2.49933390e+00,  7.64285351e-01,  2.72600150e+00],
         [ 2.78796519e-01,  1.59781364e+00, -1.20907276e+00],
         [ 7.51945351e-01, -5.86584711e-01, -5.17115959e-01],
         [-2.42664404e-01,  1.86195590e+00, -2.72468202e-03],
         [-1.98746239e+00, -4.58732454e+00,  3.97698855e+00],
         [-1.99590987e+00, -1.29950414e+00,  1.54382022e+00],
         [ 1.25695412e+00, -1.16370421e-01, -4.12536066e-01],
         [-3.25573786e+00,  4.22806449e+00,  2.99129568e+00]]),
  array([[ 0.90973817, -0.20245753,  0.06089661]]))]
```

In [38]:
```
predictions = NN.predict(inputs)

print(accuracy(predictions, targets))
```

```
0.9733333333333334
```