exercisesweek42

October 17, 2025

1 Exercises week 42

October 13-17, 2025

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

2 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.

[357]:
```
import autograd.numpy as np  # We need to use this numpy wrapper to make␣
 ↪automatic differentiation work later
from autograd import grad, elementwise_grad
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score


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

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


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


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

3 Exercise 1 - Understand the feed forward pass

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

4 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?

Answer: gradients have the same shape as the parameters they're computed with respect to, which is why we can update the parameters by subtracting the (scaled) gradient when performing gradient descent.

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.

[358]:
```
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!

[359]:
```
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)
```

```
[[0.01637783 0.02541244]
 [0.0187602  0.02910902]
 [0.00633264 0.00982596]] [0.08244799 0.09444113 0.03187928]
```

5 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?

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

We can compute their product once for reuse

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.

[360]:
```
z = W @ x + b
a = sigmoid(z)

predict = a
```

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

print(mse_der(predict, target))

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

```
[0.38319367 0.41932593 0.18960159]
[0.38319367 0.41932593 0.18960159]
```

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.

[361]:
```
def sigmoid_der(z):
    return sigmoid(z) * (1 - sigmoid(z))

print(sigmoid_der(z))

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

```
[0.2151601  0.22522131 0.16813825]
[0.2151601  0.22522131 0.16813825]
```

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

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

[362]:
```
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.

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

[363]:
```
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)
```

```
[[0.01637783 0.02541244]
 [0.0187602  0.02910902]
 [0.00633264 0.00982596]] [0.08244799 0.09444113 0.03187928]
```

You should get the same results as with autograd.

[364]:
```
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g, b_g)
```

```
[[0.01637783 0.02541244]
 [0.0187602  0.02910902]
 [0.00633264 0.00982596]] [0.08244799 0.09444113 0.03187928]
```

6 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.

[365]:
```
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)]
```

[366]:
```
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.

[367]:
```
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)

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

[368]:
```
dz2_da1 = W2
```

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.

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

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

[369]:
```
dC_da1 = dC_dz2 @ dz2_da1
dC_dz1 = dC_da1 * sigmoid_der(z1)
dC_dW1 = np.outer(dC_dz1, x)
dC_db1 = dC_dz1
```

[370]:
```
print(dC_dW1, dC_db1)
print(dC_dW2, dC_db2)
```

```
[[0.0034067  0.00516753]
 [0.00559564 0.00848788]
 [0.00340756 0.00516883]] [0.00571113 0.00938075 0.00571257]
[[0.03528078 0.03156448 0.0308783 ]
 [0.00309381 0.00276792 0.00270775]
 [0.00258347 0.00231134 0.00226109]
 [0.01269279 0.0113558  0.01110894]] [0.04188214 0.00367269 0.00306686
0.01506774]
```

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

[371]:
```
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
```

[372]:
```
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)
```

[372]:
```
[(array([[0.0034067 , 0.00516753],
         [0.00559564, 0.00848788],
         [0.00340756, 0.00516883]]),
  array([0.00571113, 0.00938075, 0.00571257])),
 (array([[0.03528078, 0.03156448, 0.0308783 ],
         [0.00309381, 0.00276792, 0.00270775],
         [0.00258347, 0.00231134, 0.00226109],
         [0.01269279, 0.0113558 , 0.01110894]]),
  array([0.04188214, 0.00367269, 0.00306686, 0.01506774]))]
```

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?

7 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.

[373]:
```
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.

[374]:
```
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.

[375]:
```
def backpropagation(
    input, layers, activation_funcs, target, activation_ders, cost_der=mse_der
):
    layer_inputs, zs, predict = feed_forward_saver(input, layers,␣
 ↪activation_funcs)

    layer_grads = [() for layer in layers]

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

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

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

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads
```

[376]:
```
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)
```

[377]:
```
layer_grads = backpropagation(x, layers, activation_funcs, target,␣
 ↪activation_ders)
print(layer_grads)
```

```
[(array([[-0.03951669, -0.00758885],
       [ 0.07398602,  0.01420839],
       [ 0.0157487 ,  0.00302441]]), array([-0.06170997,  0.1155379 ,
0.02459346])), (array([[-0.08827607, -0.04504794, -0.08517681],
       [-0.05507695, -0.02810618, -0.05314327],
       [-0.        , -0.        , -0.        ],
       [ 0.2741562 ,  0.13990395,  0.26453092]]), array([-0.1836202 ,
-0.11456378, -0.        ,  0.57026343]))]
```

[378]:
```
cost_grad = grad(cost, 0)
cost_grad(layers, x, [sigmoid, ReLU], target)
```

[378]:
```
[(array([[-0.03951669, -0.00758885],
         [ 0.07398602,  0.01420839],
         [ 0.0157487 ,  0.00302441]]),
```

```
 array([-0.06170997,  0.1155379 ,  0.02459346])),
(array([[-0.08827607, -0.04504794, -0.08517681],
        [-0.05507695, -0.02810618, -0.05314327],
        [ 0.        ,  0.        ,  0.        ],
        [ 0.2741562 ,  0.13990395,  0.26453092]]),
 array([-0.1836202 , -0.11456378,  0.        ,  0.57026343]))]
```

8 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.

[379]:
```
def create_layers_batch(network_input_size, layer_output_sizes):
    layers = []

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

        i_size = layer_output_size
    return layers
```

[380]:
```
def feed_forward_batch(input, layers, activation_funcs):
    a = input
    for (W, b), activation_func in zip(layers, activation_funcs):
        z = a @ W + b
        a = activation_func(z)
    return a
```

[381]:
```
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
```

[382]:
```
def backpropagation_batch(
    input, layers, activation_funcs, target, activation_ders, cost_der=mse_der
):
    layer_inputs, zs, predict = feed_forward_saver_batch(input, layers,␣
 ↪activation_funcs)

    layer_grads = [() for layer in layers]

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

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

        # Compute gradients for weights and biases
        dC_dW = layer_input.T @ dC_dz / input.shape[0]  # Average over batch
        dC_db = np.mean(dC_dz, axis=0)  # Average over batch

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads
```

9 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)

[383]:
```
iris = datasets.load_iris()
inputs = iris.data

# Since each prediction is a vector with a score for each of the three types of␣
 ↪flowers,
# we need to make each target a vector with a 1 for the correct flower and a 0␣
 ↪for the others.
```

```
targets = np.zeros((len(iris.data), 3))
for i, t in enumerate(iris.target):
    targets[i, t] = 1


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)
```

[384]:
```
def cross_entropy(predict, target):
    return np.sum(-target * np.log(predict))

def softmax(z):
    """Compute softmax values for each set of scores in the rows of the matrix␣
 ↪z.
    Used with batched input data."""
    e_z = np.exp(z - np.max(z, axis=0))
    return e_z / np.sum(e_z, axis=1)[:, np.newaxis]
```

A little "trick" to avoid computing the separate derivatives and still be compliant with the original interface of the backpropagation function:

[385]:
```
def cross_entropy_der(predict, target):
    """Cross entropy derivative for softmax output layer"""
    return predict - target # product of softmax and cross-entropy derivatives

def softmax_der(z):
    """Assumed to be combined with cross-entropy loss"""
    return 1
```

[386]:
```
network_input_size = 4
layer_output_sizes = [8, 3]
activation_funcs = [sigmoid, softmax]
activation_ders = [sigmoid_der, softmax_der]
layers = create_layers_batch(network_input_size, layer_output_sizes)
```

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

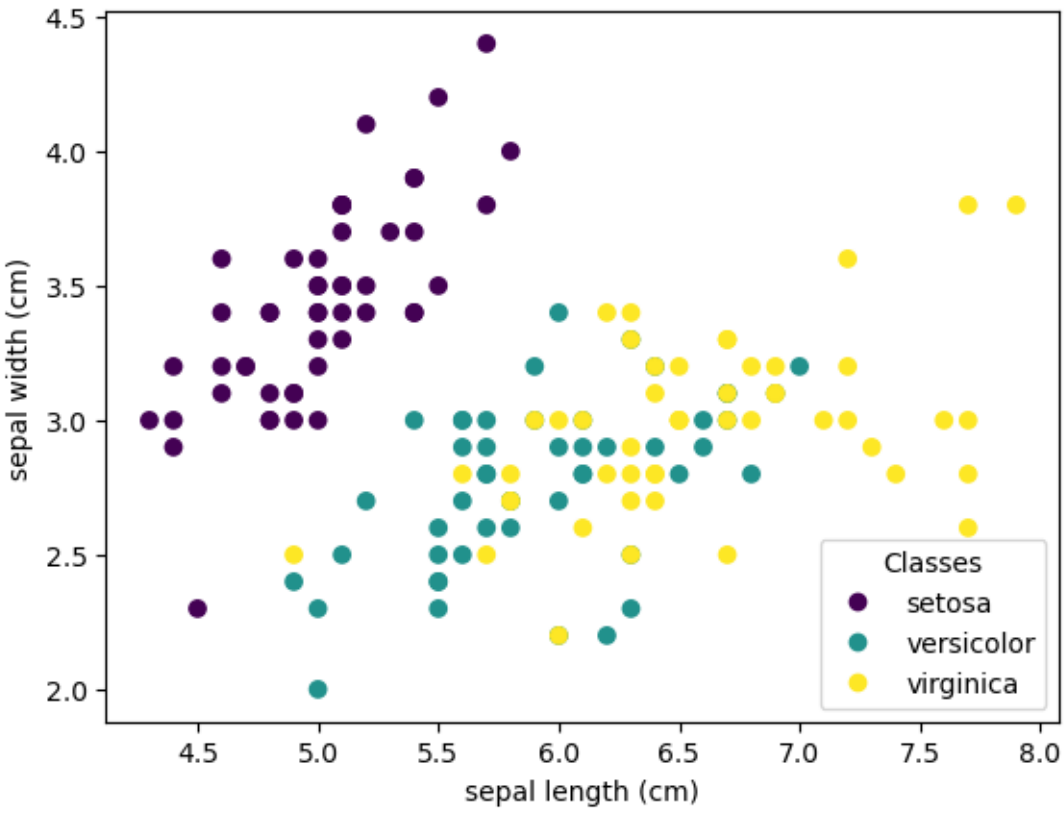
[387]:
```
def train_network(
    inputs, targets, layers, activation_funcs, activation_ders,
    learning_rate=0.01, momentum=0.9, epochs=100
):
    momentum_W = [np.zeros_like(W) for W, b in layers]
    momentum_b = [np.zeros_like(b) for W, b in layers]
```

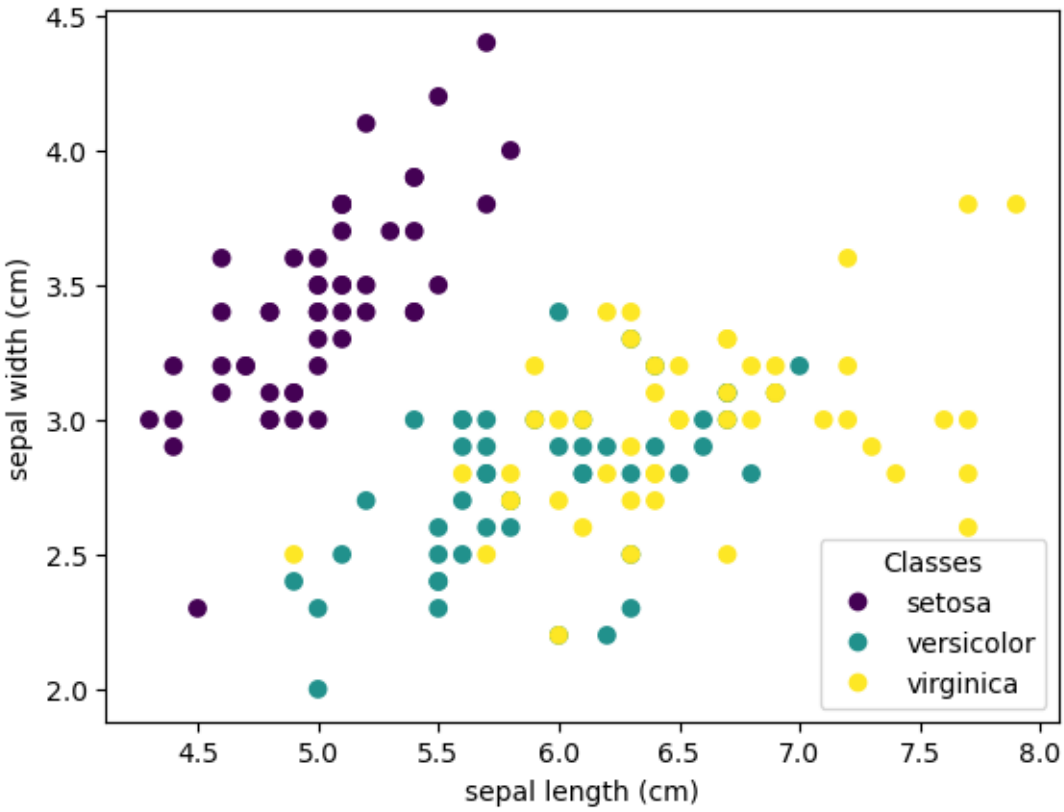
```
    for epoch in range(epochs):
        layers_grad = backpropagation_batch(
            inputs, layers, activation_funcs, targets, activation_ders,
            cost_der=cross_entropy_der
        )

        for j, ((W, b), (W_g, b_g)) in enumerate(zip(layers, layers_grad)):
            momentum_W[j] = momentum * momentum_W[j] - learning_rate * W_g
            momentum_b[j] = momentum * momentum_b[j] - learning_rate * b_g

            W += momentum_W[j]
            b += momentum_b[j]

        if epoch % 20 == 0:
            predict = feed_forward_batch(inputs, layers, activation_funcs)
            loss = cross_entropy(predict, targets)
            acc = accuracy(predict, targets)
            print(f"Epoch {epoch}: Loss = {loss:.4f}, Accuracy = {acc:.4f}")

    return layers
```

[388]:
```
print("Initial accuracy:", accuracy(feed_forward_batch(inputs, layers,␣
 ↪activation_funcs), targets))

trained_layers = train_network(
    inputs, targets, layers, activation_funcs, activation_ders,
    learning_rate=0.1, momentum=0.9, epochs=200
)
predictions = feed_forward_batch(inputs, trained_layers, activation_funcs)
print("\nFinal accuracy:", accuracy(predictions, targets))
```

```
Initial accuracy: 0.32666666666666666
Epoch 0: Loss = 202.9409, Accuracy = 0.3267
Epoch 20: Loss = 70.1576, Accuracy = 0.9200
Epoch 40: Loss = 43.8245, Accuracy = 0.9467
Epoch 60: Loss = 27.9334, Accuracy = 0.9600
Epoch 80: Loss = 20.1854, Accuracy = 0.9667
Epoch 100: Loss = 16.4243, Accuracy = 0.9733
Epoch 120: Loss = 14.3306, Accuracy = 0.9800
Epoch 140: Loss = 12.9849, Accuracy = 0.9800
Epoch 160: Loss = 12.0678, Accuracy = 0.9800
Epoch 180: Loss = 11.4054, Accuracy = 0.9800

Final accuracy: 0.98
```

[389]:
```
_, 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="Classes"
)
plt.show()

_, ax = plt.subplots()
scatter = ax.scatter(iris.data[:, 0], iris.data[:, 1], c=np.argmax(predictions,␣
 ↪axis=1))
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="Classes"
)
plt.show()
```





10 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.

[390]:
```
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 have! The first one has a layers parameter so that you can use␣
↪autograd on it
    def autograd_compliant_predict(self, layers, inputs):
        pass

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