week43

October 24, 2024

Week 43

[48]:
```
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 pltfound
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

last weeks exercise completed… check

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 will have the same shapes as the weights (10,4) The shape of the gradient of cost will have the same shape as the bias

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.

[49]:
```
def feed_forward_one_layer(W, b, x):
    z = np.dot(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!

[50]:
```
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)
```

```
[[-0.00592191 -0.00054843]
 [ 0.01333464  0.00123493]
 [ 0.0096954   0.0008979 ]] [-0.02961008  0.06667436  0.04847782]
```

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.

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

a) The intermediary results $\frac{dC}{da}$ and $\frac{da}{dz}$ can be reused for both weight gradient and bias gradient.
b) The derivative of the cost with respect to acitvation output is $\frac{dC}{da} = 2/n(a - target)$

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

      predict = a


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


      print(mse_der(predict, target))

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

```
[-0.12417357  0.33060537  0.2223989 ]
[-0.12417357  0.33060537  0.2223989 ]
```

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.

$$\sigma(z) = 1/(1 + e{-}z)$$
$$\frac{d\sigma}{dz} = \sigma * (1 - \sigma)$$

```
[52]: def sigmoid_der(z):
          sig = sigmoid(z)
          return sig * (1 - sig)


      print(sigmoid_der(z))

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

```
[0.23845722 0.20167357 0.21797688]
[0.23845722 0.20167357 0.21797688]
```

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

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

[53]:
```
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

[54]:
```
dA_dz = sigmoid_der(z)  # Derivative of the activation function
dC_dz = dC_da * dA_dz   # Chain rule

# Calculate dC/dW and dC/db
# dC/dW = dC/dz * dz/dW (where dz/dW = x.T)
dC_dW = np.outer(dC_dz, x)  # Shape (3,) x (2,) -> (3, 2)

# dC/db = dC/dz (as the bias is added directly)
dC_db = dC_dz  # Shape (3,)


print(dC_dW, dC_db)
```

```
[[-0.00592191 -0.00054843]
 [ 0.01333464  0.00123493]
 [ 0.0096954   0.0008979 ]] [-0.02961008  0.06667436  0.04847782]
```

[55]:
```
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g, b_g)
```

```
[[-0.00592191 -0.00054843]
 [ 0.01333464  0.00123493]
 [ 0.0096954   0.0008979 ]] [-0.02961008  0.06667436  0.04847782]
```

Exercise 4 - Gradient with two layers writing backpropagation by hand

[56]:
```
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)]

z1 = W1 @ x + b1
a1 = sigmoid(z1)
```

```
z2 = W2 @ a1 + b2
a2 = sigmoid(z2)
```

[57]:
```
dC_da2 = 2 * (a2 - target) / target.size

dA_dz2 = sigmoid_der(z2)
dC_dz2 = dC_da2 * dA_dz2

dC_dW2 = np.outer(dC_dz2, a1)
dC_db2 = dC_dz2

print("Gradient with respect to W2:\n", dC_dW2)
print("Gradient with respect to b2:\n", dC_db2)
```

```
Gradient with respect to W2:
 [[0.00160399 0.00180104 0.00153336]
 [0.02479391 0.02783989 0.02370222]
 [0.01660961 0.01865013 0.01587828]
 [0.02952328 0.03315028 0.02822336]]
Gradient with respect to b2:
 [0.00238876 0.03692466 0.02473607 0.04396794]
```

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.

$$\frac{dC}{d_a 1} = \frac{dC}{dz_2}\frac{dz_2}{da_1}$$

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

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

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.

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

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

[58]:
```
dC_da1 = dC_dz2 @ W2
dC_dz1 = dC_da1 * sigmoid_der(z1)
dC_dW1 = np.outer(dC_dz1, x)
dC_db1 = dC_dz1
print(dC_dW1, dC_db1)
print(dC_dW2, dC_db2)
```

```
[[0.00677529 0.00746185]
 [0.00260275 0.00286649]
 [0.00946187 0.01042066]] [0.01264914 0.0048592  0.01766484]
[[0.00160399 0.00180104 0.00153336]
 [0.02479391 0.02783989 0.02370222]
 [0.01660961 0.01865013 0.01587828]
 [0.02952328 0.03315028 0.02822336]] [0.00238876 0.03692466 0.02473607
0.04396794]
```

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

[59]:
```
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
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)
```

[59]:
```
[(array([[0.00677529, 0.00746185],
         [0.00260275, 0.00286649],
         [0.00946187, 0.01042066]]),
  array([0.01264914, 0.0048592 , 0.01766484])),
 (array([[0.00160399, 0.00180104, 0.00153336],
         [0.02479391, 0.02783989, 0.02370222],
         [0.01660961, 0.01865013, 0.01587828],
         [0.02952328, 0.03315028, 0.02822336]]),
  array([0.00238876, 0.03692466, 0.02473607, 0.04396794]))]
```

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?

By using the chain rule iteratively, you can compute gradients for any layer in a feedforward neural network. Its important to ensure that you keep track of the necessary derivatives and correctly apply the relationships between the layers. THe method can be extended to networks with any number of layers

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

[60]:
```python
def create_layers(network_input_size, layer_output_sizes):
    layers = []

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

        i_size = layer_output_size
    return layers


def feed_forward(input, layers, activation_funcs):
    a = input
    for (W, b), activation_func in zip(layers, activation_funcs):
        z = W @ a + b
        a = activation_func(z)
    return a


def cost(layers, input, activation_funcs, target):
    predict = feed_forward(input, layers, activation_funcs)
    return mse(predict, target)
```

[61]:
```python
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)
        print(a.shape)

        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

[62]:
```python
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 = 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
```

[63]:
```
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)
```

[64]:
```
layer_grads = backpropagation(x, layers, activation_funcs, target,␣
↪activation_ders)

cost_grad = grad(cost, 0)
grads = cost_grad(layers, x, activation_funcs, target)
```

```
(3,)
(4,)
```

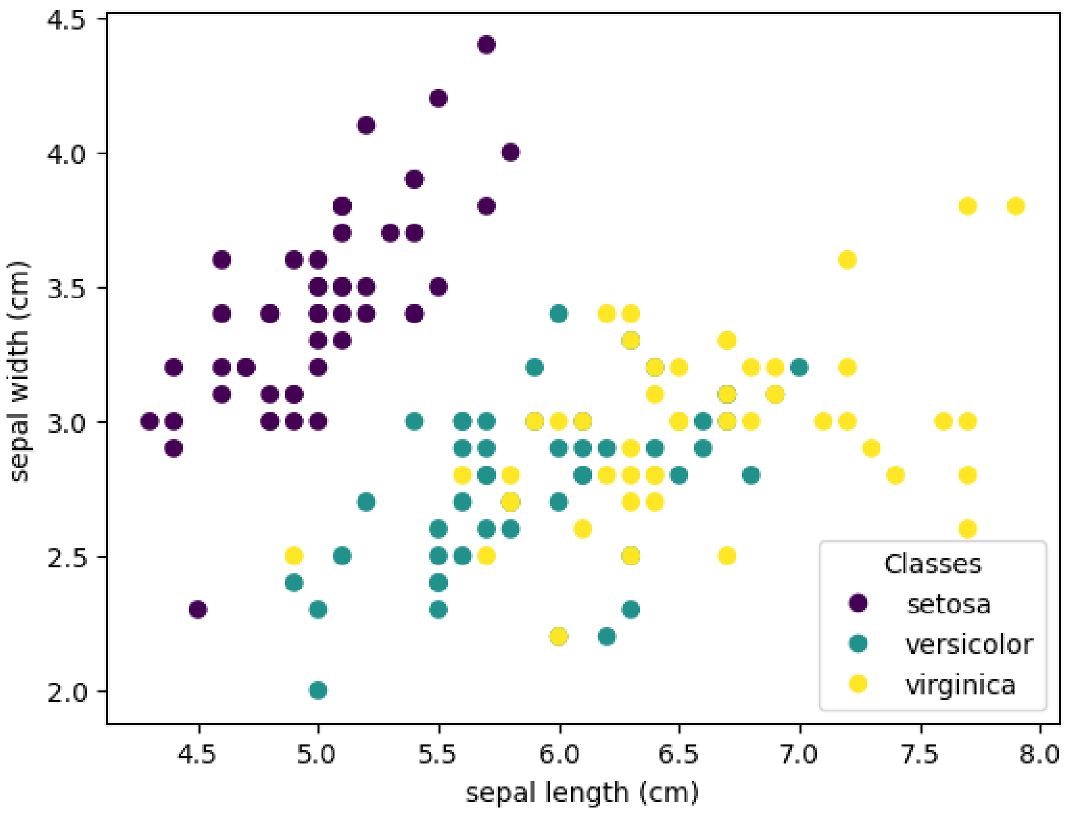
Exercise 6 - Batched inputs

[65]:
```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(layer_output_size, i_size).T
        b = np.random.randn(layer_output_size)
        layers.append((W, b))

        i_size = layer_output_size
    return layers

def feed_forward_batch(input, layers, activation_funcs):
    a = input
    for (W, b), activation_func in zip(layers, activation_funcs):
        z = a@W+b#np.einsum("ijk,ik->ij", W, a) + 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):
        print(a.shape, W.shape, b.shape)
        #(20,) (2, 30, 20) (2, 30)
        layer_inputs.append(a)
        z = np.dot(a,W) +b#np.einsum("ijk,i->ij", W, a) + b
        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_saver_batch(input, layers,␣
 ↪activation_funcs)

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

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

        dC_dz = dC_da * activation_der(z)
        dC_dW = np.outer(dC_dz,layer_input )#np.einsum("ij,ik->ijk", dC_dz,␣
 ↪layer_input)
        dC_db = dC_dz

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads

network_input_size = 20
layer_output_sizes = [30, 40]
activation_funcs = [sigmoid, ReLU]
activation_ders = [sigmoid_der, ReLU_der]


layers = create_layers_batch(network_input_size, layer_output_sizes)

x = np.random.rand(network_input_size)
target = np.random.rand(40)

layer_grads = backpropagation_batch(x, layers, activation_funcs, target,␣
 ↪activation_ders)
print(layer_grads)

cost_grad = grad(cost_batch, 0)
grads = cost_grad(layers, x, activation_funcs, target)
```

```
(20,) (20, 30) (30,)
(30,) (30, 40) (40,)
[(array([[ 2.75604058e-01,  2.67603952e-01,  3.78656826e-01,
         1.14577566e-01,  1.15774763e-01,  3.64776169e-01,
         3.17610296e-01,  8.08053318e-02,  2.26512548e-01,
  2.96335004e-01,  1.40281748e-01,  1.72605157e-01,
  2.28638306e-01,  1.85609208e-01,  2.99227234e-01,
  2.35980984e-03,  1.72704605e-01,  1.34140578e-01,
  2.42056061e-01,  3.08333105e-01],
[ 9.17846670e-02,  8.91203845e-02,  1.26104423e-01,
  3.81578698e-02,  3.85565734e-02,  1.21481735e-01,
  1.05774042e-01,  2.69106723e-02,  7.54356775e-02,
  9.86887127e-02,  4.67181566e-02,  5.74828504e-02,
  7.61436207e-02,  6.18136014e-02,  9.96519147e-02,
  7.85889591e-04,  5.75159697e-02,  4.46729572e-02,
  8.06121475e-02,  1.02684451e-01],
[-9.01973024e-02, -8.75790972e-02, -1.23923517e-01,
 -3.74979507e-02, -3.78897590e-02, -1.19380777e-01,
 -1.03944739e-01, -2.64452672e-02, -7.41310595e-02,
 -9.69819464e-02, -4.59101921e-02, -5.64887166e-02,
 -7.48267592e-02, -6.07445696e-02, -9.79284904e-02,
 -7.72298068e-04, -5.65212630e-02, -4.39003633e-02,
 -7.92180054e-02, -1.00908581e-01],
[ 1.35567558e-01,  1.31632366e-01,  1.86258438e-01,
  5.63598409e-02,  5.69487331e-02,  1.79430647e-01,
  1.56230110e-01,  3.97475335e-02,  1.11419815e-01,
  1.45764954e-01,  6.90035341e-02,  8.49031751e-02,
  1.12465459e-01,  9.12997701e-02,  1.47187620e-01,
  1.16077267e-03,  8.49520929e-02,  6.59827388e-02,
  1.19065551e-01,  1.51666730e-01],
[ 2.73829546e-02,  2.65880950e-02,  3.76218796e-02,
  1.13839844e-02,  1.15029333e-02,  3.62427511e-02,
  3.15565322e-02,  8.02850564e-03,  2.25054118e-02,
  2.94427013e-02,  1.39378526e-02,  1.71493816e-02,
  2.27166190e-02,  1.84414140e-02,  2.97300622e-02,
  2.34461590e-04,  1.71592624e-02,  1.33276897e-02,
  2.40497552e-02,  3.06347863e-02],
[-2.25529620e-02, -2.18983051e-02, -3.09858754e-02,
 -9.37599944e-03, -9.47396730e-03, -2.98500071e-02,
 -2.59903754e-02, -6.61238296e-03, -1.85357535e-02,
 -2.42493965e-02, -1.14793989e-02, -1.41244565e-02,
 -1.87097065e-02, -1.51885914e-02, -2.44860707e-02,
 -1.93105653e-04, -1.41325945e-02, -1.09768607e-02,
 -1.98076951e-02, -2.52312134e-02],
[ 6.34040789e-02,  6.15636149e-02,  8.71118788e-02,
  2.63591367e-02,  2.66345578e-02,  8.39185649e-02,
  7.30678221e-02,  1.85896669e-02,  5.21103339e-02,
  6.81733357e-02,  3.22725111e-02,  3.97086714e-02,
  5.25993751e-02,  4.27003179e-02,  6.88387077e-02,
  5.42885944e-04,  3.97315499e-02,  3.08597045e-02,
  5.56861960e-02,  7.09335584e-02],
[-5.76563538e-02, -5.59827321e-02, -7.92149873e-02,
 -2.39696206e-02, -2.42200741e-02, -7.63111546e-02,
 -6.64440566e-02, -1.69044710e-02, -4.73864127e-02,
 -6.19932666e-02, -2.93469340e-02, -3.61089893e-02,
 -4.78311212e-02, -3.88294362e-02, -6.25983210e-02,
 -4.93672089e-04, -3.61297938e-02, -2.80622015e-02,
 -5.06381147e-02, -6.45032687e-02],
[ 1.04176311e-01,  1.01152329e-01,  1.43129501e-01,
  4.33094789e-02,  4.37620106e-02,  1.37882714e-01,
  1.20054360e-01,  3.05438222e-02,  8.56200804e-02,
  1.12012456e-01,  5.30254709e-02,  6.52434821e-02,
  8.64236014e-02,  7.01589182e-02,  1.13105698e-01,
  8.91990795e-04,  6.52810727e-02,  5.07041537e-02,
  9.14954142e-02,  1.16547650e-01],
[ 4.24675881e-03,  4.12348588e-03,  5.83468991e-03,
  1.76551569e-03,  1.78396320e-03,  5.62080408e-03,
  4.89402926e-03,  1.24512229e-03,  3.49031204e-03,
  4.56620015e-03,  2.16158918e-03,  2.65965776e-03,
  3.52306766e-03,  2.86003605e-03,  4.61076627e-03,
  3.63621033e-05,  2.66119014e-03,  2.06696043e-03,
  3.72982067e-03,  4.75107784e-03],
[-1.12902341e-01, -1.09625064e-01, -1.55118333e-01,
 -4.69371732e-02, -4.74276099e-02, -1.49432064e-01,
 -1.30110370e-01, -3.31022379e-02, -9.27918009e-02,
 -1.21394858e-01, -5.74669974e-02, -7.07084153e-02,
 -9.36626265e-02, -7.60355787e-02, -1.22579672e-01,
 -9.66705846e-04, -7.07491546e-02, -5.49512417e-02,
 -9.91592651e-02, -1.26309930e-01],
[ 3.09196073e-02,  3.00220874e-02,  4.24809434e-02,
  1.28542859e-02,  1.29885977e-02,  4.09236933e-02,
  3.56322245e-02,  9.06542941e-03,  2.54121043e-02,
  3.32453812e-02,  1.57379996e-02,  1.93643146e-02,
  2.56505899e-02,  2.08232197e-02,  3.35698562e-02,
  2.64743539e-04,  1.93754715e-02,  1.50490309e-02,
  2.71559077e-02,  3.45914303e-02],
[-7.65544678e-02, -7.43322804e-02, -1.05179409e-01,
 -3.18261809e-02, -3.21587260e-02, -1.01323782e-01,
 -8.82225297e-02, -2.24452761e-02, -6.29183319e-02,
 -8.23128973e-02, -3.89660248e-02, -4.79444897e-02,
 -6.35088032e-02, -5.15566217e-02, -8.31162714e-02,
 -6.55483768e-04, -4.79721133e-02, -3.72601936e-02,
 -6.72358494e-02, -8.56456068e-02],
[ 2.02724559e-02,  1.96839965e-02,  2.78526516e-02,
  8.42791892e-03,  8.51598049e-03,  2.68316398e-02,
  2.33622856e-02,  5.94375328e-03,  1.66614588e-02,
  2.17973507e-02,  1.03186273e-02,  1.26962224e-02,
  1.68178220e-02,  1.36527543e-02,  2.20100929e-02,
  1.73579232e-04,  1.27035375e-02,  9.86690459e-03,
  1.78047844e-02,  2.26798884e-02],
[ 7.27663031e-02,  7.06540768e-02,  9.99747886e-02,
  3.02513177e-02,  3.05674074e-02,  9.63099510e-02,
  8.38569914e-02,  2.13346107e-02,  5.98049277e-02,
  7.82397868e-02,  3.70378589e-02,  4.55720400e-02,
  6.03661806e-02,  4.90054320e-02,  7.90034073e-02,
  6.23048294e-04,  4.55982967e-02,  3.54164377e-02,
  6.39088004e-02,  8.14075830e-02],
[-9.37072345e-02, -9.09871446e-02, -1.28745869e-01,
 -3.89571436e-02, -3.93641987e-02, -1.24026353e-01,
 -1.07989638e-01, -2.74743567e-02, -7.70157909e-02,
 -1.00755896e-01, -4.76967384e-02, -5.86869149e-02,
 -7.77385630e-02, -6.31083800e-02, -1.01739274e-01,
 -8.02351227e-04, -5.87207279e-02, -4.56086992e-02,
 -8.23006899e-02, -1.04835331e-01],
[ 5.91323297e-02,  5.74158640e-02,  8.12428542e-02,
  2.45832317e-02,  2.48400968e-02,  7.82646847e-02,
  6.81449936e-02,  1.73372176e-02,  4.85994829e-02,
  6.35802654e-02,  3.00982019e-02,  3.70333627e-02,
  4.90555757e-02,  3.98234517e-02,  6.42008090e-02,
  5.06309865e-04,  3.70546998e-02,  2.87805808e-02,
  5.19344269e-02,  6.61545224e-02],
[ 3.83358397e-03,  3.72230449e-03,  5.26702238e-03,
  1.59374548e-03,  1.61039819e-03,  5.07394589e-03,
  4.41788031e-03,  1.12398209e-03,  3.15073327e-03,
  4.12194628e-03,  1.95128426e-03,  2.40089485e-03,
  3.18030204e-03,  2.58177797e-03,  4.16217649e-03,
  3.28243686e-05,  2.40227814e-03,  1.86586211e-03,
  3.36693967e-03,  4.28883689e-03],
[-3.83987040e-05, -3.72840843e-05, -5.27565940e-05,
 -1.59635895e-05, -1.61303897e-05, -5.08226629e-05,
 -4.42512488e-05, -1.12582523e-05, -3.15589994e-05,
 -4.12870557e-05, -1.95448404e-05, -2.40483190e-05,
 -3.18551719e-05, -2.58601165e-05, -4.16900175e-05,
 -3.28781949e-07, -2.40621747e-05, -1.86892181e-05,
 -3.37246088e-05, -4.29586985e-05],
[-9.34796814e-02, -9.07661969e-02, -1.28433231e-01,
 -3.88625424e-02, -3.92686090e-02, -1.23725174e-01,
 -1.07727403e-01, -2.74076396e-02, -7.68287703e-02,
 -1.00511226e-01, -4.75809145e-02, -5.85444031e-02,
 -7.75497873e-02, -6.29551314e-02, -1.01492216e-01,
 -8.00402845e-04, -5.85781340e-02, -4.54979458e-02,
 -8.21008358e-02, -1.04580755e-01],
[ 1.21948721e-02,  1.18408851e-02,  1.67547300e-02,
  5.06980478e-03,  5.12277811e-03,  1.61405415e-02,
  1.40535556e-02,  3.57545784e-03,  1.00226810e-02,
  1.31121708e-02,  6.20715817e-03,  7.63739777e-03,
  1.01167412e-02,  8.21279839e-03,  1.32401456e-02,
  1.04416384e-04,  7.64179812e-03,  5.93542493e-03,
  1.07104472e-02,  1.36430604e-02],
[ 9.63448756e-02,  9.35482215e-02,  1.32369767e-01,
  4.00536967e-02,  4.04722095e-02,  1.27517406e-01,
  1.11029296e-01,  2.82476960e-02,  7.91836066e-02,
  1.03591940e-01,  4.90392909e-02,  6.03388153e-02,
  7.99267230e-02,  6.48847344e-02,  1.04602998e-01,
  8.24935551e-04,  6.03735800e-02,  4.68924782e-02,
  8.46172632e-02,  1.07786202e-01],
[ 2.17981545e-01,  2.11654078e-01,  2.99488333e-01,
  9.06220143e-02,  9.15689048e-02,  2.88509805e-01,
  2.51205238e-01,  6.39107827e-02,  1.79153949e-01,
  2.34378124e-01,  1.10952039e-01,  1.36517361e-01,
  1.80835259e-01,  1.46802563e-01,  2.36665655e-01,
  1.86642750e-03,  1.36596017e-01,  1.06094847e-01,
  1.91447668e-01,  2.43867696e-01],
[ 1.87915093e-02,  1.82460381e-02,  2.58179554e-02,
  7.81224129e-03,  7.89386977e-03,  2.48715307e-02,
  2.16556203e-02,  5.50954931e-03,  1.54443033e-02,
  2.02050072e-02,  9.56482932e-03,  1.17687360e-02,
  1.55892439e-02,  1.26553912e-02,  2.04022082e-02,
  1.60898896e-04,  1.17755167e-02,  9.14610596e-03,
  1.65041065e-02,  2.10230737e-02],
[-1.11633917e-03, -1.08393460e-03, -1.53375625e-03,
 -4.64098483e-04, -4.68947751e-04, -1.47753240e-03,
 -1.28648619e-03, -3.27303443e-04, -9.17493132e-04,
 -1.20031024e-03, -5.68213732e-04, -6.99140277e-04,
 -9.26103554e-04, -7.51813421e-04, -1.21202527e-03,
 -9.55845200e-06, -6.99543093e-04, -5.43338812e-04,
 -9.80452409e-04, -1.24890876e-03],
[ 7.80613406e-02,  7.57954124e-02,  1.07249725e-01,
  3.24526369e-02,  3.27917277e-02,  1.03318206e-01,
  8.99590728e-02,  2.28870815e-02,  6.41567955e-02,
  8.39331172e-02,  3.97330191e-02,  4.88882131e-02,
  6.47588895e-02,  5.25714452e-02,  8.47523046e-02,
  6.68386093e-04,  4.89163805e-02,  3.79936109e-02,
  6.85592976e-02,  8.73314266e-02],
[ 1.58999302e-02,  1.54383945e-02,  2.18451686e-02,
  6.61011787e-03,  6.67918562e-03,  2.10443768e-02,
  1.83233208e-02,  4.66175697e-03,  1.30677819e-02,
  1.70959235e-02,  8.09302310e-03,  9.95779950e-03,
  1.31904194e-02,  1.07080189e-02,  1.72627798e-02,
  1.36140273e-04,  9.96353677e-03,  7.73873159e-03,
  1.39645059e-02,  1.77881084e-02],
[ 3.93717102e-04,  3.82288466e-04,  5.40934229e-04,
  1.63680998e-04,  1.65391267e-04,  5.21104869e-04,
  4.53725560e-04,  1.15435315e-04,  3.23586905e-04,
  4.23332515e-04,  2.00400980e-04,  2.46576929e-04,
  3.26623680e-04,  2.65154005e-04,  4.27464242e-04,
  3.37113139e-06,  2.46718996e-04,  1.91627946e-04,
          3.45791755e-04,  4.40472530e-04],
        [ 5.52437216e-02,  5.36401326e-02,  7.59002333e-02,
          2.29666109e-02,  2.32065842e-02,  7.31179116e-02,
          6.36637026e-02,  1.61971028e-02,  4.54035265e-02,
          5.93991561e-02,  2.81189105e-02,  3.45980073e-02,
          4.58296262e-02,  3.72046170e-02,  5.99788920e-02,
          4.73014362e-04,  3.46179413e-02,  2.68879377e-02,
          4.85191610e-02,  6.18041271e-02],
        [-2.56999412e-02, -2.49539353e-02, -3.53095605e-02,
         -1.06843010e-02, -1.07959390e-02, -3.40151962e-02,
         -2.96170020e-02, -7.53505696e-03, -2.11221824e-02,
         -2.76330916e-02, -1.30812031e-02, -1.60953449e-02,
         -2.13204082e-02, -1.73079663e-02, -2.79027906e-02,
         -2.20051092e-04, -1.61046184e-02, -1.25085421e-02,
         -2.25716072e-02, -2.87519085e-02]]), array([ 3.97357756e-01,
1.32332411e-01, -1.30043795e-01,  1.95457284e-01,
         3.94799318e-02, -3.25161916e-02,  9.14141203e-02, -8.31272208e-02,
         1.50198315e-01,  6.12285090e-03, -1.62779246e-01,  4.45789728e-02,
        -1.10373961e-01,  2.92282256e-02,  1.04912298e-01, -1.35104311e-01,
         8.52552390e-02,  5.52714767e-03, -5.53621125e-05, -1.34776232e-01,
         1.75822050e-02,  1.38907184e-01,  3.14279326e-01,  2.70930408e-02,
        -1.60950471e-03,  1.12546526e-01,  2.29240478e-02,  5.67649640e-04,
         7.96487592e-02, -3.70534129e-02])), (array([[-0.00000000e+00,
-0.00000000e+00, -0.00000000e+00, …,
         -0.00000000e+00, -0.00000000e+00, -0.00000000e+00],
        [-0.00000000e+00, -0.00000000e+00, -0.00000000e+00, …,
         -0.00000000e+00, -0.00000000e+00, -0.00000000e+00],
        [-4.13123978e-03, -1.70571448e-03, -1.50195342e-03, …,
         -5.79147066e-05, -6.88082302e-03, -4.99404110e-04],
        …,
        [-0.00000000e+00, -0.00000000e+00, -0.00000000e+00, …,
         -0.00000000e+00, -0.00000000e+00, -0.00000000e+00],
        [ 6.80619420e-02,  2.81015497e-02,  2.47445977e-02, …,
          9.54141521e-04,  1.13361171e-01,  8.22765451e-03],
        [ 3.06509758e-02,  1.26552357e-02,  1.11434679e-02, …,
          4.29687544e-04,  5.10510044e-02,  3.70523720e-03]]), array([-0.
, -0.        , -0.00839076,  0.03652466,  0.13319049,
        0.07413218,  0.113247  ,  0.26194643, -0.02027839,  0.16732357,
        0.17573989, -0.        ,  0.03544514, -0.        , -0.        ,
        0.08770556, -0.        ,  0.11764719, -0.        , -0.        ,
        0.06314318, -0.        , -0.        , -0.        , -0.        ,
        0.04699132,  0.08239305, -0.        ,  0.13031629, -0.        ,
        0.28306272, -0.00598917, -0.        , -0.01432129, -0.        ,
        0.09824954,  0.07383394, -0.        ,  0.13823729,  0.0622537 ]))]
```

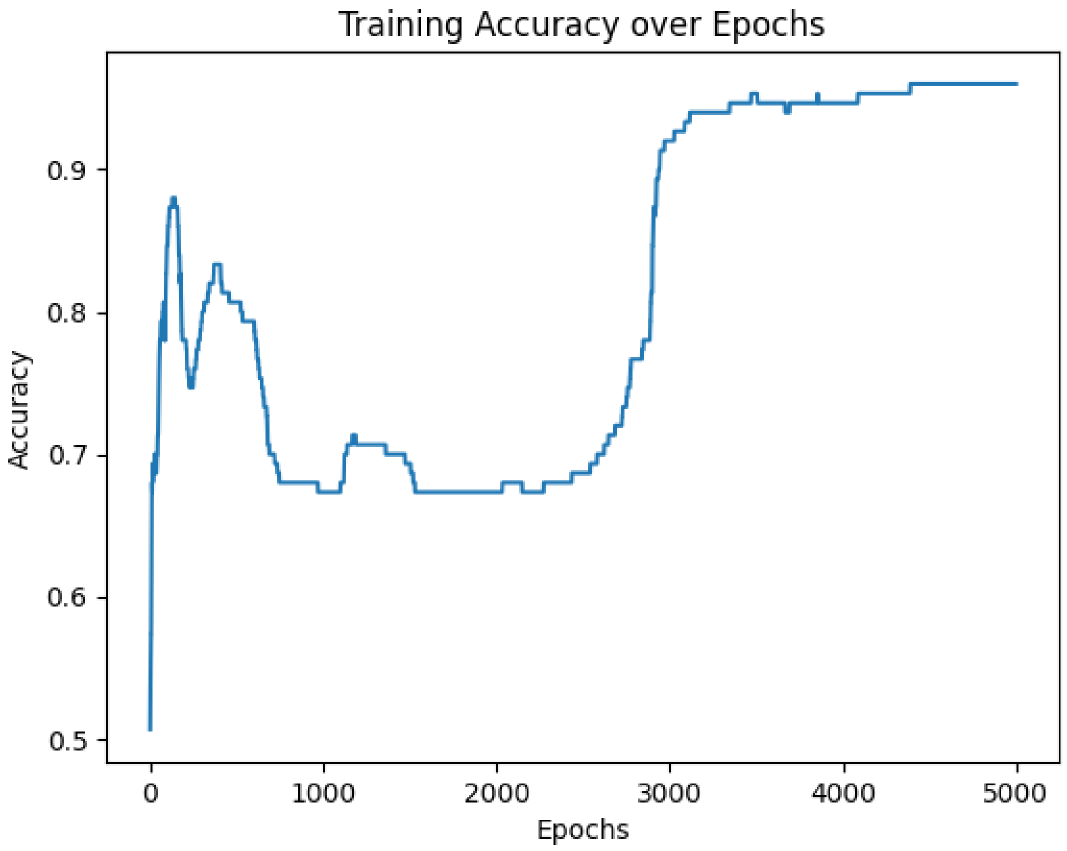
Exercise 7 - Training

[66]:
```
import autograd.numpy as np  # We need to use this numpy wrapper to make␣
  ↪automatic differentiation work later
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score

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="Classes"
)
```



[67]:
```
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)


def ReLU(z):
    return np.where(z > 0, z, 0)


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


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]


def softmax_vec(z):
    """Compute softmax values for each set of scores in the vector z.P
    Use this function when you use the activation function on one vector at a␣
↪time"""
    e_z = np.exp(z - np.max(z))
    return e_z / np.sum(e_z)

def sigmoid_der(z):
    sig = sigmoid(z)
    return sig * (1 - sig)

def ReLU_der(z):
    return np.where(z > 0, 1, 0)


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


def feed_forward_saver_batch(input, layers, activation_funcs):
    layer_inputs = []
    zs = []
    a = input
    for (W, b), activation_func in zip(layers, activation_funcs):
        print(a.shape, W.shape, b.shape)
        #(20,) (2, 30, 20) (2, 30)
        layer_inputs.append(a)
        z = np.dot(a,W) +b#np.einsum("ijk,i->ij", W, a) + b
        a = activation_func(z)

        zs.append(z)

    return layer_inputs, zs, a

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 = np.dot(a,W) + b
        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_saver_batch(input, layers,␣
 ↪activation_funcs)

    layer_grads = [() for layer in layers]

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

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

        dC_dz = dC_da * activation_der(z)
        dC_dW = np.dot(layer_input.T, dC_dz)  # (4, 150) . (150, 8) => (4, 8)
        dC_db = dC_dz.sum(axis=0) #shape 8

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads



def train_network(
    inputs, layers, activation_funcs, targets, learning_rate=0.001, epochs=100
):
    for i in range(epochs):
        layers_grad = backpropagation_batch(
            inputs, layers, activation_funcs, targets, activation_derivatives
        )
        for (W, b), (W_g, b_g) in zip(layers, layers_grad):
            W -= learning_rate * W_g
            b -= learning_rate * b_g


network_input_size = inputs.shape[1]
layer_output_sizes = [8, 3]
activation_functions = [sigmoid, softmax]
activation_derivatives = [sigmoid_der, lambda x: 1]

layers = create_layers_batch(network_input_size, layer_output_sizes)

epochs = 5000
learning_rate = 0.005
accvals = []
for epoch in range(epochs):
    train_network(inputs, layers, activation_functions, targets, learning_rate,␣
 ↪1)
    preds = feed_forward_batch(inputs, layers, activation_functions)
    acc = accuracy(preds, targets)
    accvals.append(acc)
print(f"Final accuracy: {accvals[-1]}")

plt.plot(range(1, epochs + 1), accvals)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over Epochs')
plt.show()
```

```
Final accuracy: 0.96
```



Exercise 8 (Optional) - Object orientation

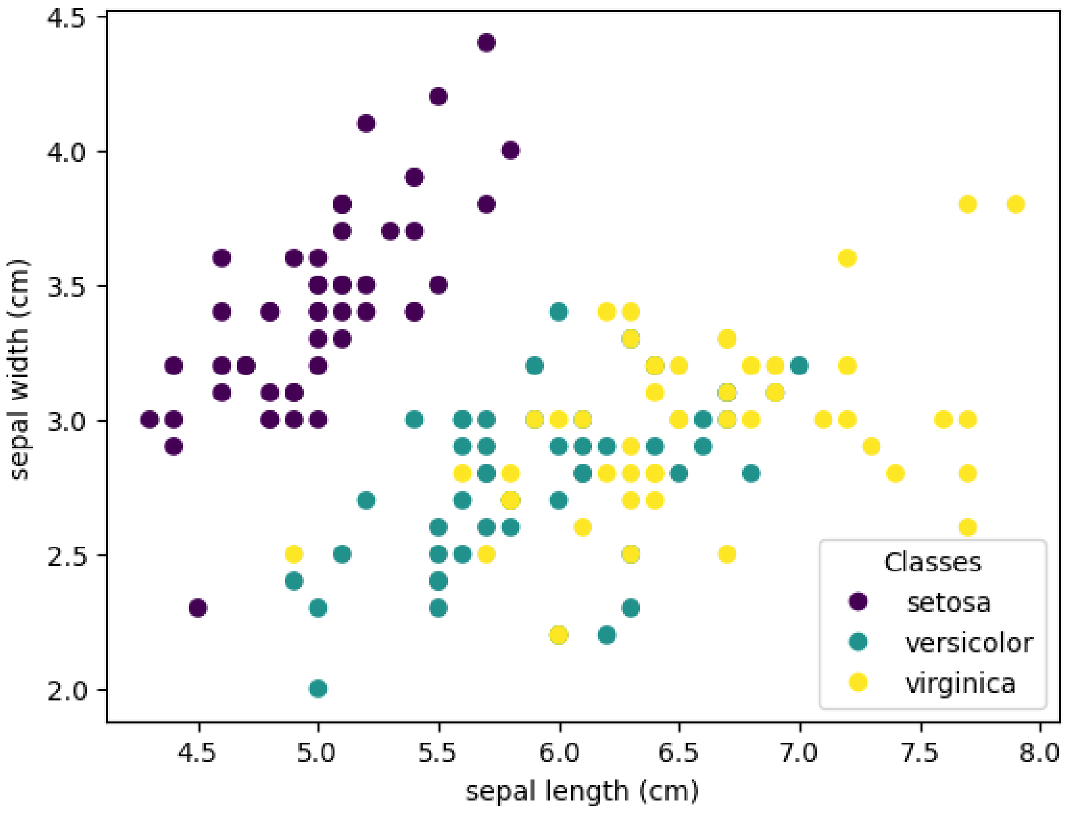
Passing in the layers, activations functions, activation derivatives and cost derivatives into the functions each time leads to code which is easy to understand in isoloation, but messier when used in a larger context with data splitting, data scaling, gradient methods and so forth. Creating an object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

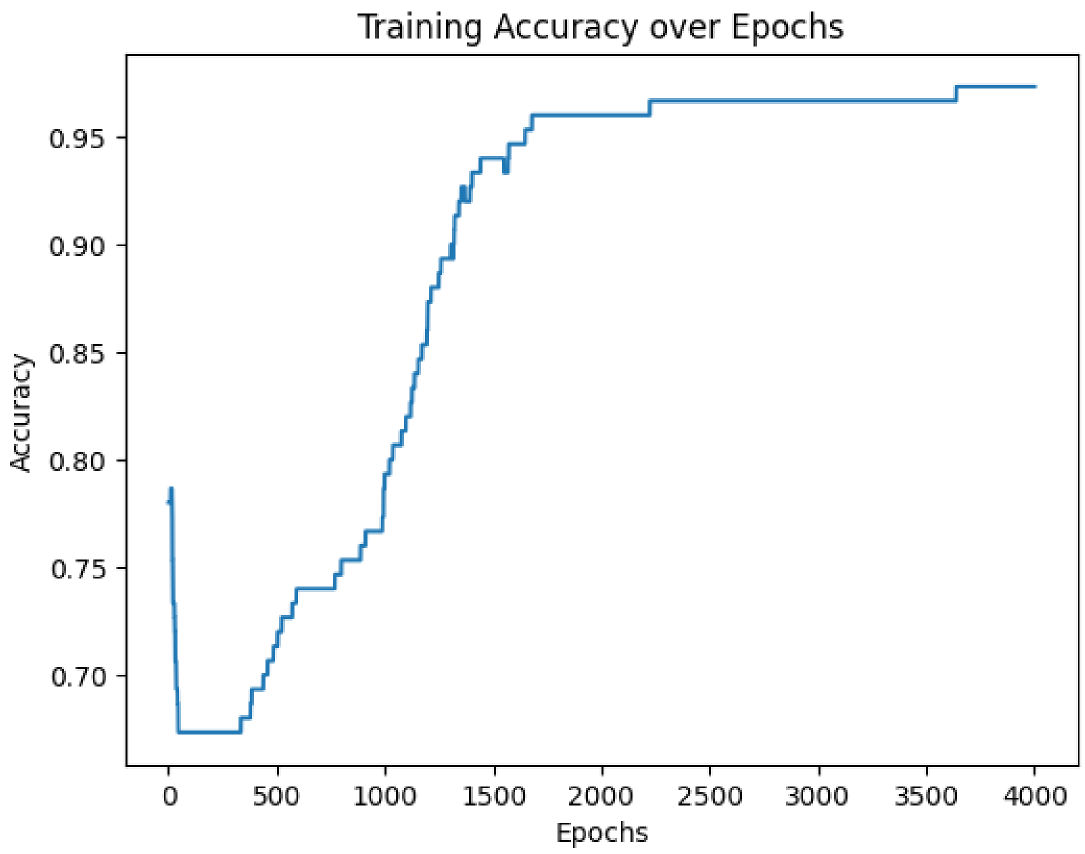
```
[68]: import autograd.numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
np.random.seed(0)

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)



def ReLU(z):
    return np.where(z > 0, z, 0)



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



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



def softmax_vec(z):
    """Compute softmax values for each set of scores in the vector z.P
    Use this function when you use the activation function on one vector at a␣
 ↪time"""
    e_z = np.exp(z - np.max(z))
    return e_z / np.sum(e_z)

def sigmoid_der(z):
    sig = sigmoid(z)
    return sig * (1 - sig)

def ReLU_der(z):
    return np.where(z > 0, 1, 0)



class NeuralNetwork:
    def __init__(
        self,
        network_input_size,
        layer_output_sizes,
        activation_funcs,
        activation_ders,
        cost_fun,
        cost_der,
    ):
        self.NIS = network_input_size
        self.LOS = layer_output_sizes
        self.AF = activation_funcs
        self.AFder = activation_ders
        self.CF = cost_fun
        self.CFder = cost_der
        self.layers = self.create_layers()

    def predict(self, inputs):
        a = inputs

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


    def cost(self, inputs, targets):
        predict = self.predict(inputs)
        return self.CF(predict, targets)

    def _feed_forward_saver(self, inputs):
        layer_inputs = []
        zs = []
        a = inputs
        for (W, b), activation_func in zip(self.layers, activation_funcs):
            layer_inputs.append(a)
            z = np.dot(a,W) + b
            a = activation_func(z)
            zs.append(z)
        return layer_inputs, zs, a

    def create_layers(self):
        layers = []

        i_size = self.NIS
        for layer_output_size in self.LOS:
            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

    def compute_gradient(self, inputs, targets):
        layer_inputs, zs, predict = self._feed_forward_saver(inputs)


        layer_grads = [() for layer in self.layers]

        for i in reversed(range(len(self.layers))):
            layer_input, z, activation_der = layer_inputs[i], zs[i],␣
↪activation_ders[i]

            if i == len(self.layers) - 1:
                dC_da = self.CFder(predict, targets)
            else:
                (W, b) = self.layers[i + 1]
                dC_da = dC_dz @ W.T

            dC_dz = dC_da * activation_der(z)
            dC_dW = np.dot(layer_input.T, dC_dz)
            dC_db = dC_dz.sum(axis=0)

            layer_grads[i] = (dC_dW, dC_db)

        return layer_grads

    def update_weights(self, layer_grads, learning_rate,inputs, targets):
        layer_grads = self.compute_gradient(inputs, targets)

        for (W, b), (W_g, b_g) in zip(self.layers, layer_grads):
            W -= learning_rate * W_g
            b -= learning_rate * b_g

    # 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, inputs):
        a = inputs
        for (W, b), activation_func in zip(self.layers, self.AF):
            z = np.dot(a,W) + b
            a = activation_func(z)
        return a

    def autograd_gradient(self, targets,inputs):
        from autograd import grad

        def cost_func_autograd(inputs, targets):
            pred = self.autograd_compliant_predict(inputs)
            return self.CF(pred, targets)

        gradients = grad(cost_func_autograd)(inputs, targets)
        return gradients




newdata = datasets.load_iris()
_, ax = plt.subplots()
scatter = ax.scatter(newdata.data[:, 0], newdata.data[:, 1], c=newdata.target)
ax.set(xlabel=newdata.feature_names[0], ylabel=newdata.feature_names[1])
_ = ax.legend(
    scatter.legend_elements()[0], newdata.target_names, loc="lower right",␣
  ↪title="Classes"
)
plt.show()
data = newdata.data
print(data.shape)
targets = np.zeros((len(data), 3))
for i, t in enumerate(newdata.target):
    targets[i, t] = 1


network_input_size = data.shape[1]
layer_output_sizes = [8,3]#[network_input_size, 3]
activation_funcs = [sigmoid, softmax]
activation_ders = [sigmoid_der, lambda x: 1]

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

def cross_entropy(predict, target):
    predict = np.clip(predict, 1e-15, 1 - 1e-15)
    return np.sum(-target * np.log(predict))
```

```
network = NeuralNetwork(
    network_input_size,
    layer_output_sizes,
    activation_funcs,
    activation_ders,
    cross_entropy,
    mse_der,
)

learning_rate = 0.01
epochs = 4000

accuracy_values = []

for epoch in range(epochs):
    grads = network.compute_gradient(data, targets)
    network.update_weights(grads, learning_rate, data, targets)
    preds = network.predict(data)
    acc = accuracy(preds, targets)
    accuracy_values.append(acc)
print(f"Final accuracy: {accuracy_values[-1]}")


plt.plot(range(1, epochs + 1), accuracy_values)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over Epochs')
plt.show()

#again, but using auto grad
network = NeuralNetwork(
    network_input_size,
    layer_output_sizes,
    activation_funcs,
    activation_ders,
    cross_entropy,
    mse_der,
)

accuracy_values = []

for epoch in range(epochs):
    grads = network.autograd_gradient(targets,data)
    network.update_weights(grads, learning_rate, data, targets)
    preds = network.predict(data)
    acc = accuracy(preds, targets)
    accuracy_values.append(acc)
```

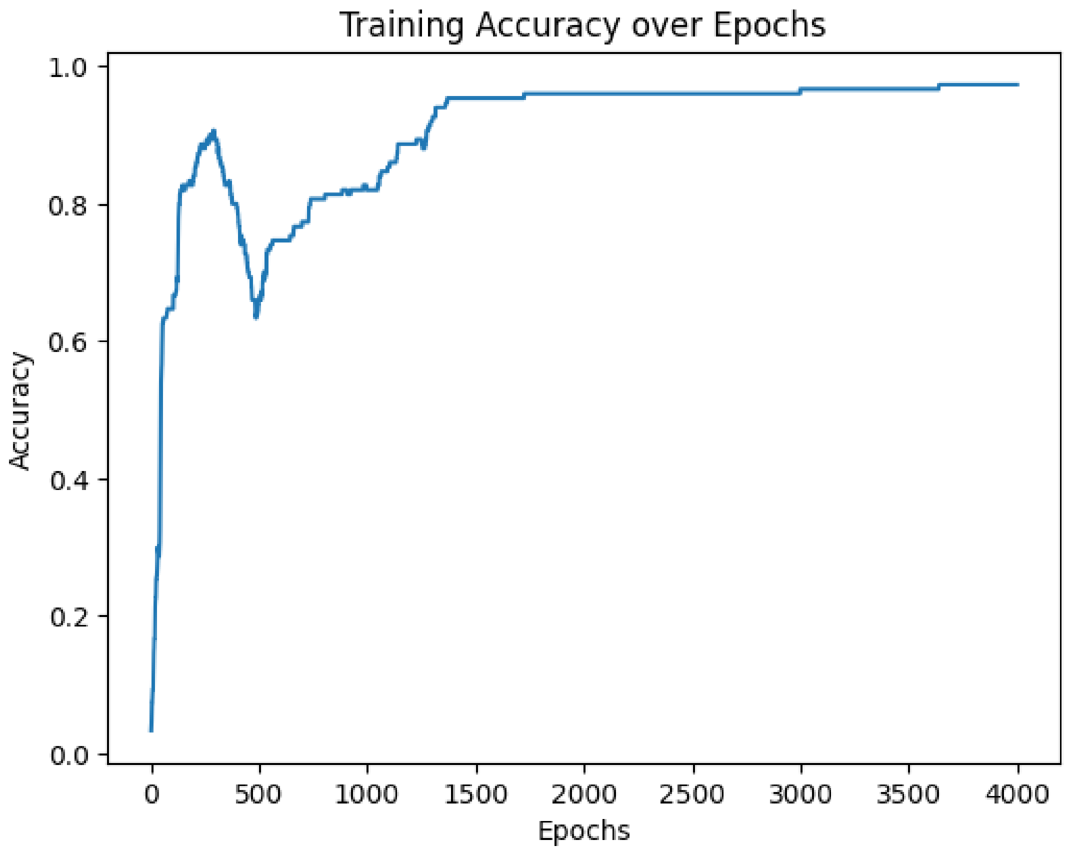
```
print(f"Final accuracy: {accuracy_values[-1]}")


plt.plot(range(1, epochs + 1), accuracy_values)
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over Epochs')
plt.show()
```



```
(150, 4)
Final accuracy: 0.9733333333333334
```



```
Final accuracy: 0.9733333333333334
```


Training Accuracy over Epochs
Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
1500
2000
2500
3000
3500
4000
Epochs