exercisesweek42

October 16, 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.

[1]:

```python
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).

ok

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?

They will be equal one to each parameter that is going to be updated, (10,4) and (10)

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

[2]:
```
def feed_forward_one_layer(W, b, x):
    z = W @ x + b
    a = sigmoid(z)
    return a


def cost_one_layer(W, b, x, target):
    predict = feed_forward_one_layer(W, b, x)
    return mse(predict, target)


x = (np.random.rand(2))
target = (np.random.rand(3))
```

```
W = np.random.rand(3, 2)
b = np.random.rand(3)
```

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

[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)
```

```
[[-0.00975426 -0.00422459]
 [ 0.00177657  0.00076944]
 [-0.01049165 -0.00454396]] [-0.01674269  0.00304939 -0.01800839]
```

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?

This calculation can be reused between the steps $\frac{dC}{da}\frac{da}{dz}$, as it will be computed once and then be used for the backpropagation from this node.

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.

[4]:

```
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))
```

```
[-0.09755788  0.02085524 -0.09237403]
[-0.09755788  0.02085524 -0.09237403]
```

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.

[5]:

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


print(sigmoid_der(z))

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

```
[0.17161799 0.14621719 0.19495075]
[0.17161799 0.14621719 0.19495075]
```

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

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

[6]:

```
dC_da = mse_der(predict, target)
dC_dz = dC_da * sigmoid_der(z)
dC_dz.shape
```

[6]:

```
(3,)
```

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.

The shape of the itermidiary z is (3,1), while the Weights for the bias will be (3,2), where 3 is the output and the 2 is the input from x.

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

[7]:

```
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)
```

```
[[-0.00975426 -0.00422459]
 [ 0.00177657  0.00076944]
 [-0.01049165 -0.00454396]] [-0.01674269  0.00304939 -0.01800839]
```

You should get the same results as with autograd.

[8]:

```
W_g, b_g = autograd_one_layer(W, b, x, target)
print(W_g, b_g)
```

```
[[-0.00975426 -0.00422459]
 [ 0.00177657  0.00076944]
 [-0.01049165 -0.00454396]] [-0.01674269  0.00304939 -0.01800839]
```

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.

[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)]
```

[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.

[11]:

```
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)

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

As $z_2 = w_1^{(2)} a_1^{(1)} + w_2^{(2)} a_2^{(1)} + b^{(2)} \implies w_1^{(2)}$

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

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

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

[12]:

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

[13]:

```
print(dC_dW1, dC_db1)
print(dC_dW2, dC_db2)
```

```
[[0.00320615 0.00037602]
 [0.00135325 0.00015871]
 [0.00217013 0.00025451]] [0.0048105  0.00203041 0.00325606]
[[0.00119309 0.00137111 0.00137299]
 [0.00326913 0.00375693 0.00376207]
 [0.00746337 0.00857702 0.00858875]
 [0.01231335 0.01415069 0.01417005]] [0.00184464 0.00505443 0.0115392  0.0190378
]
```

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

[14]:

```
def feed_forward_two_layers(layers, x):
    W1, b1 = layers[0]
    z1 = W1 @ x + b1
    a1 = sigmoid(z1)

    W2, b2 = layers[1]
    z2 = W2 @ a1 + b2
    a2 = sigmoid(z2)

    return a2
```

[15]:

```
def cost_two_layers(layers, x, target):
    predict = feed_forward_two_layers(layers, x)
    return mse(predict, target)


grad_two_layers = grad(cost_two_layers, 0)
grad_two_layers(layers, x, target)
```

[15]:

```
[(array([[0.00320615, 0.00037602],
         [0.00135325, 0.00015871],
         [0.00217013, 0.00025451]]),
  array([0.0048105 , 0.00203041, 0.00325606])),
 (array([[0.00119309, 0.00137111, 0.00137299],
         [0.00326913, 0.00375693, 0.00376207],
         [0.00746337, 0.00857702, 0.00858875],
         [0.01231335, 0.01415069, 0.01417005]]),
  array([0.00184464, 0.00505443, 0.0115392 , 0.0190378 ]))]
```

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.

[16]:

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

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

        i_size = layer_output_size
    return layers


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


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

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

Here is a function which does that for you.

[17]:

```
def feed_forward_saver(input, layers, activation_funcs):
    layer_inputs = []
    zs = []
    a = input
    for (W, b), activation_func in zip(layers, activation_funcs):
        layer_inputs.append(a)
        z = W @ a + b
        a = activation_func(z)

        zs.append(z)

    return layer_inputs, zs, a
```

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

[18]:

```
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
```

[19]:

```
network_input_size = 2
layer_output_sizes = [3, 4]
activation_funcs = [sigmoid, ReLU]
activation_ders = [sigmoid_der, ReLU_der]

layers = create_layers(network_input_size, layer_output_sizes)

x = np.random.rand(network_input_size)
target = np.random.rand(4)
```

[20]:

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

```
[(array([[-0.01936369, -0.01845224],
       [-0.00497568, -0.00474147],
       [ 0.08486317,  0.08086865]]), array([-0.03034576, -0.00779762,
0.13299308])), (array([[-0.03084204, -0.12135819, -0.13659013],
       [ 0.00166773,  0.00656223,  0.00738587],
       [-0.        , -0.        , -0.        ],
       [-0.02792314, -0.10987282, -0.1236632 ]]), array([-0.38261845,
0.02068943, -0.        , -0.34640733]))]
```

[21]:

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

[21]:

```
[(array([[-0.01936369, -0.01845224],
         [-0.00497568, -0.00474147],
         [ 0.08486317,  0.08086865]]),
  array([-0.03034576, -0.00779762,  0.13299308])),
 (array([[-0.03084204, -0.12135819, -0.13659013],
         [ 0.00166773,  0.00656223,  0.00738587],
         [ 0.        ,  0.        ,  0.        ],
         [-0.02792314, -0.10987282, -0.1236632 ]]),
  array([-0.38261845,  0.02068943,  0.        , -0.34640733]))]
```

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.

[39]:

```python
# Flipping the W shape
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.rand(i_size, layer_output_size)
        b = np.random.rand(1,layer_output_size)
        layers.append((W, b))

        i_size = layer_output_size
    return layers

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

[23]:

```python
# Flipping the W shape
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
```

[24]:

```python
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 = layer_input.T @ dC_dz
        dC_db = dC_dz

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads
```

[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(10,network_input_size)
target = np.random.rand(10,4)
```

[26]:

```
backpropagation_batch(x, layers, activation_funcs, target, activation_ders)
```

[26]:

```
[(array([[-0.01149496, -0.01249239,  0.10501532],
         [-0.00801965, -0.00838401,  0.07469544]]),
  array([[-9.20288399e-04, -4.08343768e-03,  2.67759185e-02],
         [-1.63472739e-03, -3.80081977e-03,  1.79412089e-02],
         [-3.75097445e-03, -2.73817690e-03,  2.54389510e-02],
         [-3.57285746e-03, -1.45808792e-03,  1.89239934e-02],
         [ 2.06735389e-03, -5.29524936e-03,  1.93613225e-02],
         [-5.43144887e-03,  1.97290699e-04,  3.48620680e-02],
         [-3.03528319e-03, -1.79651878e-03,  1.98259212e-02],
         [-6.82717131e-05, -5.07272190e-03,  2.71177081e-02],
         [-4.01928134e-03, -7.70158426e-04,  2.17541098e-02],
         [-1.98997771e-03, -1.67737843e-03,  2.40914951e-02]])),
 (array([[0.02090961, 0.18819985, 0.0834566 , 0.        ],
         [0.05645196, 0.50000837, 0.22105245, 0.        ],
         [0.0860295 , 0.76377037, 0.33853996, 0.        ]]),
  array([[-0.00834477,  0.10990497,  0.05017784, -0.        ],
         [ 0.008283  ,  0.11194551,  0.04288172, -0.        ],
         [ 0.00301993,  0.11900602,  0.06765257, -0.        ],
         [ 0.02251493,  0.0834819 ,  0.03890059, -0.        ],
         [ 0.00051632,  0.08848498,  0.01005778, -0.        ],
         [ 0.01830618,  0.08084403,  0.06080786, -0.        ],
         [ 0.00958851,  0.08457729,  0.04460419, -0.        ],
         [ 0.00501343,  0.11870115,  0.03633636, -0.        ],
         [ 0.02649856,  0.07637425,  0.03873511, -0.        ],
         [ 0.02455606,  0.07380525,  0.02607315, -0.        ]]))]
```

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)

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

[36]:
```
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 der_softmax(predict, target):
    """Compute the derivative of the softmax function for each set of scores in␣
 ↪the rows of the matrix z.
    Used with batched input data."""
    return predict - target

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


def cost_batch(layers, input, activation_funcs, target):
    _, _, predict = feed_forward_saver_batch(input, layers, activation_funcs)
    return cross_entropy(predict, target)
```

[ ]:

```

```

[34]:

```
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_dz = 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 = layer_input.T @ dC_dz
        dC_db = dC_dz

        layer_grads[i] = (dC_dW, dC_db)

    return layer_grads
```

[47]:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

iterations = 10000
batch_size = 100
learning_rate = 0.01
momentum = 0.9

iris = datasets.load_iris()
scaler = StandardScaler()
input = iris.data
```

```
input_scaled = scaler.fit_transform(input)
targets = np.zeros((len(iris.data), 3))
for i, t in enumerate(iris.target):
    targets[i, t] = 1
network_input_size = 4
layer_output_sizes = [8,3]
activation_funcs = [sigmoid, softmax]
layers = create_layers_batch(network_input_size, layer_output_sizes)


X_train, X_test, y_train, y_test = train_test_split(input_scaled, targets,␣
 ↪test_size=0.2)

velocities = []
for W, b in layers:
    v_W = np.zeros_like(W)
    v_b = np.zeros_like(b)
    velocities.append((v_W, v_b))



for i in range(iterations):
    for j in range(0, len(X_train), batch_size):
        x_batch = X_train[j : j + batch_size]
        y_batch = y_train[j : j + batch_size]

        layer_grads = backpropagation_batch(
            x_batch, layers, activation_funcs, y_batch, [sigmoid_der,␣
 ↪der_softmax], cost_der=der_softmax
        )

        # Update weights and biases using gradient descent
        for k in range(len(layers)):
            W, b = layers[k]
            dC_dW, dC_db = layer_grads[k]
            v_W, v_b = velocities[k]

            v_W = momentum * v_W + learning_rate * dC_dW
            v_b = momentum * v_b + learning_rate * np.mean(dC_db, axis=0,␣
 ↪keepdims=True)

            W -= v_W
            b -= v_b
            layers[k] = (W, b)
            velocities[k] = (v_W, v_b)

    if i % 1000 == 0:
        train_cost = cost_batch(layers, X_train, activation_funcs, y_train)
        test_cost = cost_batch(layers, X_test, activation_funcs, y_test)
        print(f"Iteration {i}, Train Cost: {train_cost}, Test Cost:␣
 ↪{test_cost}")
```

```
Iteration 0, Train Cost: 118.33133617913381, Test Cost: 27.978061246115068
Iteration 1000, Train Cost: 0.21208921077655513, Test Cost: 13.4562755037409
Iteration 2000, Train Cost: 0.06928534659030888, Test Cost: 16.576224347157343
Iteration 3000, Train Cost: 0.04609350820910309, Test Cost: 17.412816273041624
Iteration 4000, Train Cost: 0.03747667679196304, Test Cost: 18.005386973661796
Iteration 5000, Train Cost: 0.0419666463384467, Test Cost: 18.255483063215213
Iteration 6000, Train Cost: 0.05361972971657784, Test Cost: 18.311312776098664
Iteration 7000, Train Cost: 0.06391088986906733, Test Cost: 18.302759760614293
Iteration 8000, Train Cost: 0.07261844735603776, Test Cost: 18.30705771877406
Iteration 9000, Train Cost: 0.07764351504553943, Test Cost: 18.358348176887073
```

[48]:

```
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)

_ , _ ,predictions = feed_forward_saver_batch(X_test, layers, activation_funcs)
print("Test Accuracy:", accuracy(predictions, y_test))
```

```
Test Accuracy: 0.9666666666666667
```

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.

[31]:

```
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
```