Notebook_W41

October 10, 2025

1 Exercises week 41

October 6-10, 2025

Date: **Deadline is Friday October 10 at midnight**

2 Overarching aims of the exercises this week

This week, you will implement the entire feed-forward pass of a neural network! Next week you will compute the gradient of the network by implementing back-propagation manually, and by using autograd which does back-propagation for you (much easier!). Next week, you will also use the gradient to optimize the network with a gradient method! However, there is an optional exercise this week to get started on training the network and getting good results!

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 pieces of the feed-forward pass 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/1zKibVQf-iAYaAn2-GlKfgRjHtLnPlBX4#offline=true&sandboxMode=true), an updated link will be provided on the course discord (you can also send an email to k.h.fredly@fys.uio.no if you encounter any trouble), though we recommend that you set up VSCode and your python environment to run code like this locally.

First, here are some functions you are going to need, don't change this cell. If you are unable to import autograd, just swap in normal numpy until you want to do the final optional exercise.

[35]:
```
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
from numpy.typing import NDArray


# Defining some activation functions
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.
    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)
```

3 Exercise 1

In this exercise you will compute the activation of the first 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!

[36]:
```
np.random.seed(2024)

x = np.random.randn(2)  # network input. This is a single input with two␣
 ↪features
W1 = np.random.randn(4, 2)  # first layer weights

print(x)
print(W1)
```

```
[1.66804732 0.73734773]
[[-0.20153776 -0.15091195]
 [ 0.91605181  1.16032964]
 [-2.619962   -1.32529457]
 [ 0.45998862  0.10205165]]
```

a) Given the shape of the first layer weight matrix, what is the input shape of the neural network? What is the output shape of the first layer?

Answer: The first layer weight matrix has the shape (4, 2), the input is given by x (2,). The output shape of the first layer is given by the multiplication of W1 and x, which results in the

shape (4,)

b) Define the bias of the first layer, `b1`with the correct shape. (Run the next cell right after the previous to get the random generated values to line up with the test solution below)

[37]:
```
b1 = np.random.randn(4)

print(b1)
```

```
[ 1.05355278  1.62404261 -1.50063502 -0.27783169]
```

c) Compute the intermediary `z1` for the first layer

[38]:
```
z1 = W1 @ x + b1

print(z1)
```

```
[ 0.60610368  4.0076268  -6.84805855  0.56469864]
```

d) Compute the activation `a1` for the first layer using the ReLU activation function defined earlier.

[39]:
```
a1 = ReLU(z1)

print(a1)
```

```
[0.60610368 4.0076268  0.         0.56469864]
```

Confirm that you got the correct activation with the test below. Make sure that you define `b1` with the randn function right after you define `W1`.

[40]:
```
sol1 = np.array([0.60610368, 4.0076268, 0.0, 0.56469864])

print(np.allclose(a1, sol1))
```

```
True
```

4 Exercise 2

Now we will add a layer to the network with an output of length 8 and ReLU activation.

a) What is the input of the second layer? What is its shape?

b) Define the weight and bias of the second layer with the right shapes.

[41]:
```
W2 = np.random.randn(8, 4)
b2 = np.random.randn(8)

print(W2)
print(b2)
```

```
[[ 1.19399502  0.86181533 -0.41704604 -0.24953642]
 [ 0.94367735 -0.76631064  0.20822873  1.40872293]
 [-1.48910401 -1.47580853  0.99084632 -0.88323043]
```

```
 [-0.36618388 -1.53470503 -0.35157551  0.63991811]
 [ 0.68923917  0.75725237 -1.43054977 -0.4288793 ]
 [-0.68501186 -0.12564086  1.14458724  0.32720979]
 [-0.13672744  0.17919391  0.96897707  0.00587009]
 [ 0.59050544 -0.39247637  0.03591147 -0.33075495]]
[ 0.80877607  0.05331094 -1.31890201 -1.0797807  -0.37596827  0.14872677
  1.81491884  0.55249894]
```

c) Compute the intermediary `z2` and activation `a2` for the second layer.

[42]:
```
z2 = W2 @ a1 + b2
a2 = ReLU(z2)

print(z2)
print(a2)
```

```
[ 4.84538217 -1.65030586 -8.63470228 -7.09089022  2.83437944 -0.58520821
  2.45350499 -0.84926925]
[4.84538217 0.         0.         0.         2.83437944 0.
 2.45350499 0.        ]
```

Confirm that you got the correct activation shape with the test below.

[43]:
```
print(
    np.allclose(np.exp(len(a2)), 2980.9579870417283)
)  # This should evaluate to True if a2 has the correct shape :)
```

```
True
```

5 Exercise 3

We often want our neural networks to have many layers of varying sizes. To avoid writing very long and error-prone code where we explicitly define and evaluate each layer we should keep all our layers in a single variable which is easy to create and use.

a) Complete the function below so that it returns a list `layers` of weight and bias tuples `(W, b)` for each layer, in order, with the correct shapes that we can use later as our network parameters.

[44]:
```
def create_layers(network_input_size, layer_output_sizes):
    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␣
 ↪size
        W = np.random.randn(layer_output_size, i_size)
```

```
        # b has the shape of the current output layer size
        b = np.random.randn(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
```

b) Comple the function below so that it evaluates the intermediary `z` and activation `a` for each layer, with ReLU actication, and returns the final activation `a`. This is the complete feed-forward pass, a full neural network!

[45]:
```
def feed_forward_all_relu(layers, input):

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

    # For each weight and bias in the network layers
    for W, b in layers:

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

        # Calculate a using the ReLU function
        a = ReLU(z)

        print("z", z.shape)
        print("a", a.shape)

    return a
```

c) Create a network with input size 8 and layers with output sizes 10, 16, 6, 2. Evaluate it and make sure that you get the correct size vectors along the way.

[46]:
```
input_size = 10
layer_output_sizes = [10, 16, 6, 2]

x = np.random.rand(input_size)
layers = create_layers(input_size, layer_output_sizes)
predict = feed_forward_all_relu(layers, x)
print(predict)
```

```
z (10,)
a (10,)
z (16,)
```

```
a (16,)
z (6,)
a (6,)
z (2,)
a (2,)
[8.22965745 0.        ]
```

d) Why is a neural network with no activation functions always mathematically equivelent to a neural network with only one layer?

Answer:

6 Exercise 4 - Custom activation for each layer

So far, every layer has used the same activation, ReLU. We often want to use other types of activation however, so we need to update our code to support multiple types of activation functions. Make sure that you have completed every previous exercise before trying this one.

a) Complete the `feed_forward` function which accepts a list of activation functions as an argument, and which evaluates these activation functions at each layer.

[47]:
```python
def feed_forward(input, layers, activation_funcs):

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

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

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

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

b) You are now given a list with three activation functions, two ReLU and one sigmoid. (Don't call them yet! you can make a list with function names as elements, and then call these elements of the list later. If you add other functions than the ones defined at the start of the notebook, make sure everything is defined using autograd's numpy wrapper, like above, since we want to use automatic differentiation on all of these functions later.)

Evaluate a network with three layers and these activation functions.

[48]:
```python
input_size = 10
layer_output_sizes = [10, 16, 6, 2]

# ReLU hidden layers, sigmoid output layer
activation_funcs = [ReLU, ReLU, sigmoid]
layers = create_layers(input_size, layer_output_sizes)
```

```
x = np.random.randn(input_size)
feed_forward(x, layers, activation_funcs)
```

[48]:
```
array([1.00000000e+00, 3.08415905e-18, 9.99999999e-01, 3.81169268e-17,
       1.09675871e-03, 1.00000000e+00])
```

[49]:
```
# Sigmoid hidden layers, ReLU output layer
activation_funcs = [sigmoid, sigmoid, ReLU]
layers = create_layers(input_size, layer_output_sizes)

x = np.random.randn(input_size)
feed_forward(x, layers, activation_funcs)
```

[49]:
```
array([0.7421218 , 3.32184119, 2.18902417, 0.        , 2.89282724,
       0.        ])
```

c) How does the output of the network change if you use sigmoid in the hidden layers and ReLU in the output layer?

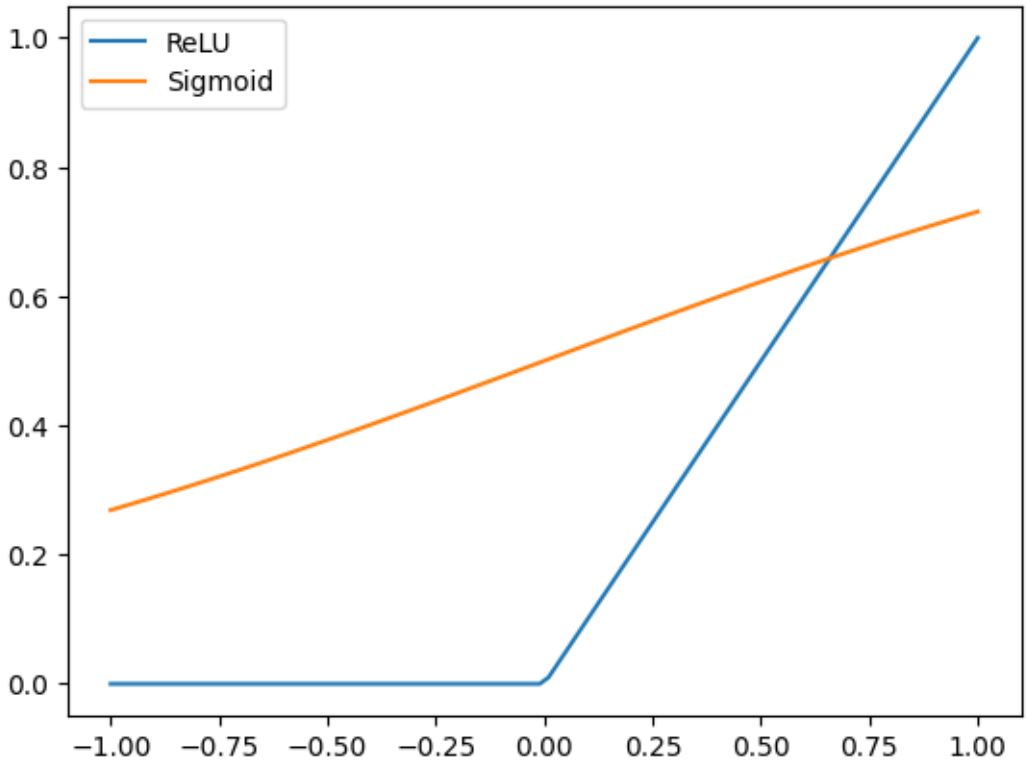
Answer: The output values are significantly higher. This makes sense because ReLU will set all values lower than 0 to 0, while sigmoid always returns a positive number, as seen in the plot below.

[50]:
```
x = np.linspace(-1, 1, 100)

y_ReLU = ReLU(x)
y_sigmoid = sigmoid(x)

plt.plot(x, y_ReLU, label="ReLU")
plt.plot(x, y_sigmoid, label="Sigmoid")
plt.legend()
```

[50]:
```
<matplotlib.legend.Legend at 0x7e485f473020>
```



7 Exercise 5 - Processing multiple inputs at once

So far, the feed forward function has taken one input vector as an input. This vector then undergoes a linear transformation and then an element-wise non-linear operation for each layer. This approach of sending one vector in at a time is great for interpreting how the network transforms data with its linear and non-linear operations, but not the best for numerical efficiency. Now, we want to be able to send many inputs through the network at once. This will make the code a bit harder to understand, but it will make it faster, and more compact. It will be worth the trouble.

To process multiple inputs at once, while still performing the same operations, you will only need to flip a couple things around.

a) Complete the function `create_layers_batch` so that the weight matrix is the transpose of what it was when you only sent in one input at a time.

[51]:
```
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␣
↪size
        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
```

b) Make a matrix of inputs with the shape (number of features, number of inputs), you choose the number of inputs and features per input. Then complete the function `feed_forward_batch` so that you can process this matrix of inputs with only one matrix multiplication and one broadcasted vector addition per layer. (Hint: You will only need to swap two variable around from your previous implementation, but remember to test that you get the same results for equivelent inputs!)

[52]:
```
def feed_forward_batch(inputs : NDArray, layers : list, activation_funcs :␣
↪list):
    # 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):

        # 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)

    return a
```

c) Create and evaluate a neural network with 4 inputs and layers with output sizes 12, 10, 3 and activations ReLU, ReLU, softmax.

[53]:
```
inputs = np.random.randn(12, 4) # (n_features, n_inputs)
network_input_size = 4
layer_output_sizes = [10, 16, 6, 2]
activation_funcs = [ReLU, ReLU, softmax]
layers = create_layers_batch(network_input_size, layer_output_sizes)
```

```
feed_forward_batch(inputs, layers, activation_funcs)
```

[53]:
```
array([[5.72153441e-04, 7.43029751e-15, 3.76666309e-06, 9.99418676e-01,
        5.66787142e-10, 5.40286256e-06],
       [3.44958004e-04, 2.36852071e-04, 6.15408601e-10, 9.55704691e-01,
        1.48851716e-09, 4.37134969e-02],
       [7.02536694e-01, 1.30067188e-19, 1.25317712e-02, 2.82421922e-01,
        1.57234908e-03, 9.37263471e-04],
       [2.81708100e-03, 3.88785355e-19, 4.75635872e-01, 2.98492239e-03,
        5.11961006e-01, 6.60111933e-03],
       [1.00701037e-02, 2.94273322e-10, 5.09831479e-01, 2.59853783e-04,
        2.43876561e-01, 2.35962002e-01],
       [7.78025374e-04, 9.99220784e-01, 3.04734909e-13, 1.05138753e-07,
        1.09455573e-13, 1.08543015e-06],
       [3.04060671e-05, 4.63402917e-20, 4.89394948e-01, 1.59304273e-04,
        4.89394948e-01, 2.10203930e-02],
       [7.66735437e-04, 1.62223561e-15, 4.26532899e-01, 1.14270800e-03,
        7.59611703e-02, 4.95596488e-01],
       [1.31361204e-01, 1.36856007e-09, 2.41706789e-06, 3.76919552e-01,
        1.02105978e-05, 4.91706616e-01],
       [8.36664887e-02, 1.53857298e-10, 6.59565618e-01, 5.95643695e-02,
        6.15310509e-02, 1.35672473e-01],
       [1.27770624e-04, 8.93049998e-12, 2.63900893e-01, 7.99579923e-05,
        2.07166277e-03, 7.33819716e-01],
       [9.99938556e-01, 4.74011950e-11, 1.55634057e-05, 4.19713670e-05,
        4.39953447e-08, 3.86516910e-06]])
```

You should use this batched approach moving forward, as it will lead to much more compact code. However, remember that each input is still treated separately, and that you will need to keep in mind the transposed weight matrix and other details when implementing backpropagation.

8 Exercise 6 - Predicting on real data

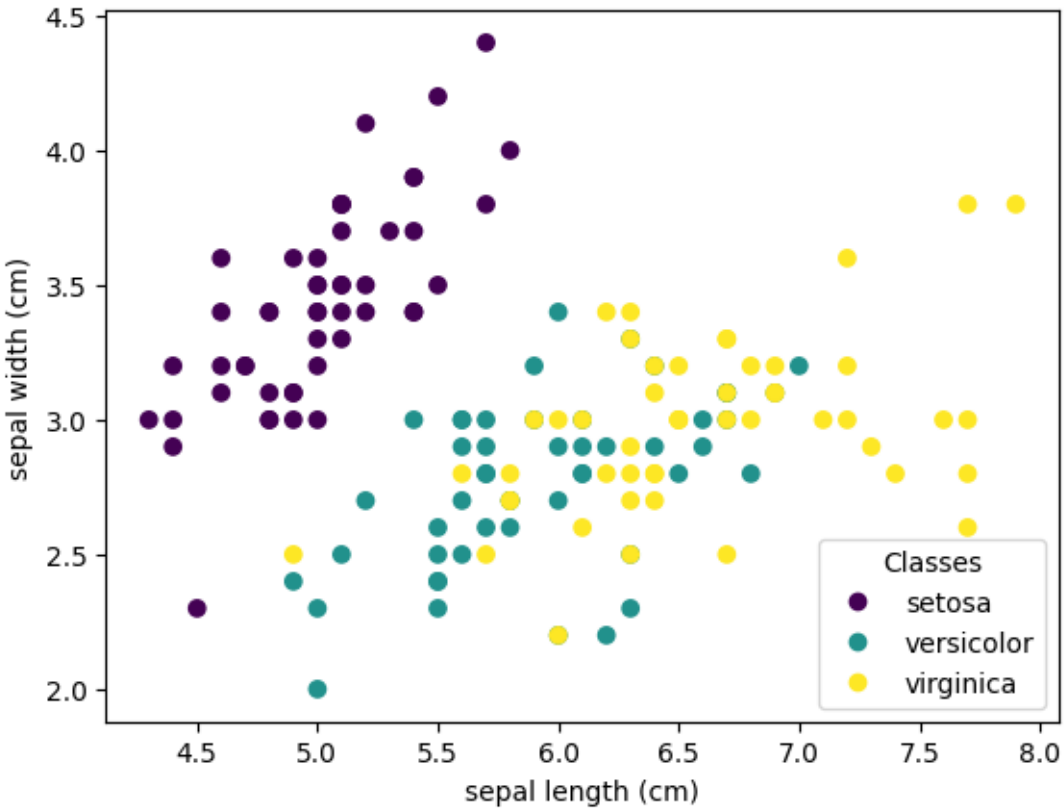
You will now evaluate your neural network on the iris data set (https://scikit-learn.org/1.5/auto_examples/datasets/plot_iris_dataset.html).

This dataset contains data on 150 flowers of 3 different types which can be separated pretty well using the four features given for each flower, which includes the width and length of their leaves. You will later train your network to actually make good predictions.

[54]:
```
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"
)
```



[55]:
```
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)
```

a) What should the input size for the network be with this dataset? What should the output size of the last layer be?

Answer: The input size should be 4, as the dataset contains petal and sepal length and widths in a (150x4) array. The output size should be 3 because we are trying to predict one of three irises'.

b) Create a network with two hidden layers, the first with sigmoid activation and the last with softmax, the first layer should have 8 "nodes", the second has the number of nodes you found in exercise a). Softmax returns a "probability distribution", in the sense that the numbers in the output are positive and add up to 1 and, their magnitude are in some sense relative to their magnitude before going through the softmax function. Remember to use the batched version of the create_layers and feed forward functions.

[56]:
```
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)
```

c) Evaluate your model on the entire iris dataset! For later purposes, we will split the data into train and test sets, and compute gradients on smaller batches of the training data. But for now, evaluate the network on the whole thing at once.

[57]:
```
predictions = feed_forward_batch(inputs, layers, activation_funcs)
```

d) Compute the accuracy of your model using the accuracy function defined above. Recreate your model a couple times and see how the accuracy changes.

[58]:
```
print(accuracy(predictions, targets))
```

```
0.20666666666666667
```

9 Exercise 7 - Training on real data (Optional)

To be able to actually do anything useful with your neural network, you need to train it. For this, we need a cost function and a way to take the gradient of the cost function wrt. the network parameters. The following exercises guide you through taking the gradient using autograd, and updating the network parameters using the gradient. Feel free to implement gradient methods like ADAM if you finish everything.

Since we are doing a classification task with multiple output classes, we use the cross-entropy loss function, which can evaluate performance on classification tasks. It sees if your prediction is "most certain" on the correct target.

[59]:
```
def cross_entropy(predict, target):
    return np.sum(-target * np.log(predict))
```

```
def cost(input, layers, activation_funcs, target):
    predict = feed_forward_batch(input, layers, activation_funcs)
    return cross_entropy(predict, target)
```

To improve our network on whatever prediction task we have given it, we need to use a sensible cost function, take the gradient of that cost function with respect to our network parameters, the weights and biases, and then update the weights and biases using these gradients. To clarify, we need to find and use these

$$\frac{\partial C}{\partial W}, \frac{\partial C}{\partial b}$$

Now we need to compute these gradients. This is pretty hard to do for a neural network, we will use most of next week to do this, but we can also use autograd to just do it for us, which is what we always do in practice. With the code cell below, we create a function which takes all of these gradients for us.

[60]:
```
from autograd import grad

gradient_func = grad(cost, 1)  # Taking the gradient wrt. the second input to␣
 ↪the cost function, i.e. the layers
```

a) What shape should the gradient of the cost function wrt. weights and biases be?

b) Use the `gradient_func` function to take the gradient of the cross entropy wrt. the weights and biases of the network. Check the shapes of what's inside. What does the `grad` func from autograd actually do?

[61]:
```
layers_grad = gradient_func(inputs, layers, activation_funcs, targets)  # Don't␣
 ↪change this
```

c) Finish the `train_network` function.

[62]:
```
def train_network(inputs, layers, activation_funcs, targets, learning_rate=0.
 ↪001, epochs=100):
    for i in range(epochs):
        layers_grad = gradient_func(inputs, layers, activation_funcs, targets)
        for (W, b), (W_g, b_g) in zip(layers, layers_grad):
            W -= W_g * learning_rate
            b -= b_g * learning_rate

    return layers
```

e) What do we call the gradient method used above?

Answer: Here we use the classic gradient descent method with w -= w * η

d) Train your network and see how the accuracy changes! Make a plot if you want.

[63]:
```
layers = train_network(inputs, layers, activation_funcs, targets)
```

[64]:
```
predictions = feed_forward_batch(inputs, layers, activation_funcs)
```

[65]:
```
print(accuracy(predictions, targets))
```

```
0.7066666666666667
```

e) How high of an accuracy is it possible to acheive with a neural network on this dataset, if we use the whole thing as training data?

Answer: By calculating the accuracy after every 50 epochs, it seems like the maximum accuracy for we can achieve for this dataset is ≈ 97, even when we increase the number of epochs substantially.

[66]:
```
iris = datasets.load_iris()


# 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

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)
```

[67]:
```
def train_network(inputs, layers, activation_funcs, targets, learning_rate=0.
 ↪001, epochs=100):
    acc = []
    for i in range(epochs):
        layers_grad = gradient_func(inputs, layers, activation_funcs, targets)
        for (W, b), (W_g, b_g) in zip(layers, layers_grad):
            W -= W_g * learning_rate
            b -= b_g * learning_rate


        if i % 50 == 0:
            predictions = feed_forward_batch(inputs, layers, activation_funcs)
            current_acc = accuracy(predictions, targets)

            print(f"Accuracy after epoch {i} is {current_acc}")

            acc.append(current_acc)
```

```
    return layers, acc
```

[68]:
```
layers, acc = train_network(inputs, layers, activation_funcs, targets,␣
↪epochs=1500)
```

```
Accuracy after epoch 0 is 0.21333333333333335
Accuracy after epoch 50 is 0.6933333333333334
Accuracy after epoch 100 is 0.9133333333333333
Accuracy after epoch 150 is 0.96
Accuracy after epoch 200 is 0.96
Accuracy after epoch 250 is 0.9666666666666667
Accuracy after epoch 300 is 0.9666666666666667
Accuracy after epoch 350 is 0.9666666666666667
Accuracy after epoch 400 is 0.9666666666666667
Accuracy after epoch 450 is 0.9666666666666667
Accuracy after epoch 500 is 0.9666666666666667
Accuracy after epoch 550 is 0.9666666666666667
Accuracy after epoch 600 is 0.9666666666666667
Accuracy after epoch 650 is 0.9666666666666667
Accuracy after epoch 700 is 0.9666666666666667
Accuracy after epoch 750 is 0.9666666666666667
Accuracy after epoch 800 is 0.9733333333333334
Accuracy after epoch 850 is 0.9733333333333334
Accuracy after epoch 900 is 0.9733333333333334
Accuracy after epoch 950 is 0.9733333333333334
Accuracy after epoch 1000 is 0.9733333333333334
Accuracy after epoch 1050 is 0.9733333333333334
Accuracy after epoch 1100 is 0.9733333333333334
Accuracy after epoch 1150 is 0.9733333333333334
Accuracy after epoch 1200 is 0.9733333333333334
Accuracy after epoch 1250 is 0.9733333333333334
Accuracy after epoch 1300 is 0.9733333333333334
Accuracy after epoch 1350 is 0.9733333333333334
Accuracy after epoch 1400 is 0.9733333333333334
Accuracy after epoch 1450 is 0.9733333333333334
```

[ ]:
```

```