# Exercises week 41

In [ ]:
```
import autograd.numpy as np  # We need to use this numpy wrapper to make automat
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)


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 ti
    e_z = np.exp(z - np.max(z))
    return e_z / np.sum(e_z)
```

# Exercise 1

**a)** The first weight matrix has shape $W_1 \in \mathbb{R}^{4\times 2}$.
Therefore:

- the network input is $x \in \mathbb{R}^2$ → input shape = (2,)
- the first layer output is $z_1 = W_1 x + b_1 \in \mathbb{R}^4$ → output shape = (4,)

**b)** Bias vector: $b_1 \in \mathbb{R}^4$

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

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

b1 = np.random.randn(4)  # first layer bias

print("Input x:", x)
print("Shape of x:", x.shape)
print("Weights W1:", W1)
print("Shape of W1:", W1.shape)
print("Bias b1:", b1)
print("Shape of b1:", b1.shape)
```

```
Input x: [1.66804732 0.73734773]
Shape of x: (2,)
Weights W1: [[-0.20153776 -0.15091195]
 [ 0.91605181  1.16032964]
 [-2.619962   -1.32529457]
 [ 0.45998862  0.10205165]]
Shape of W1: (4, 2)
Bias b1: [ 1.05355278  1.62404261 -1.50063502 -0.27783169]
Shape of b1: (4,)
```

**c)** Computing the intermediary $z_1$ for the first layer

In [ ]:
```
z1 = W1 @ x + b1  # first layer pre-activation
print("Intermediary z1:", z1)
print("Shape of z1:", z1.shape)
```

```
Intermediary z1: [ 0.60610368  4.0076268  -6.84805855  0.56469864]
Shape of z1: (4,)
```

**d)** Computing the activation a1 for the first layer using the ReLU activation function defined earlier

In [ ]:
```
a1 = ReLU(z1)  # first layer activation

print("Activation a1:", a1)
print("Shape of a1:", a1.shape)
```

```
Activation a1: [0.60610368 4.0076268  0.         0.56469864]
Shape of a1: (4,)
```

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

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

```
True
```

# Exercise 2

**a)** The input to the second layer is $a_1$ from the first layer, shape (4,)

**b)** With 8 output nodes:

$$W_2 \in \mathbb{R}^{8\times 4}, \qquad b_2 \in \mathbb{R}^{8}$$

In [ ]:
```
w2=np.random.randn(8, 4)  # second layer weights
b2=np.random.randn(8)  # second layer bias

print("Weights w2:", w2)
print("Shape of w2:", w2.shape)
print("Bias b2:", b2)
print("Shape of b2:", b2.shape)
```

```
Weights w2: [[ 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]]
Shape of w2: (8, 4)
Bias b2: [ 0.80877607  0.05331094 -1.31890201 -1.0797807  -0.37596827  0.14872677
  1.81491884  0.55249894]
Shape of b2: (8,)
```

**c)** Forward pass:

$$z_2 = W_2 a_1 + b_2, \qquad a_2 = \mathrm{ReLU}(z_2)$$

In [ ]:
```
z2= w2 @ a1 + b2  # second layer pre-activation
a2=ReLU(z2)  # second layer activation

print("Intermediary z2:", z2)
print("Shape of z2:", z2.shape)
print("Activation a2:", a2)
print("Shape di a2:", a2.shape)
print(np.exp(len(a2)))
```

```
Intermediary z2: [ 4.84538217 -1.65030586 -8.63470228 -7.09089022  2.83437944 -0.
58520821
  2.45350499 -0.84926925]
Shape of z2: (8,)
Activation a2: [4.84538217 0.         0.         0.         2.83437944 0.
 2.45350499 0.        ]
Shape di a2: (8,)
2980.9579870417283
```

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

```
True
```

# Exercise 3

**a)** Function that returns a list of weight and bias tuples (W, b) for each layer

In [ ]:
```
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
```

**b)** Function that evaluates the intermediary z and activation a for each layer, with ReLU activation, and returns the final activation a

In [ ]:
```
def feed_forward_all_relu(layers, input):
    a = input
    for idx, (W, b) in enumerate(layers):
        z = W @ a + b
        a = ReLU(z)
        print(f"Layer {idx+1}: z.shape={z.shape}, a.shape={a.shape}")
    return a
```

**c)** Creation of a network with input size 8 and layers with output sizes 10, 16, 6, 2

In [ ]:
```
input_size = 8
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)
print(predict.shape)
```

```
Layer 1: z.shape=(10,), a.shape=(10,)
Layer 2: z.shape=(16,), a.shape=(16,)
Layer 3: z.shape=(6,), a.shape=(6,)
Layer 4: z.shape=(2,), a.shape=(2,)
[5.36337158 0.        ]
(2,)
```

**d)** A network without activation functions is mathematically equivalent to a single linear layer, because a composition of affine transformations is still affine:

$$W_2(W_1x + b_1) + b_2 = (W_2W_1)x + (W_2b_1 + b_2)$$

Hence the entire network reduces to one layer with $\tilde{W} = W_2W_1$ and $\tilde{b} = W_2b_1 + b_2$.

# Exercise 4: Custom activation for each layer

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

In [ ]:
```
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
```

**b)** Evaluate a network with three layers and three activation functions

In [ ]:
```
network_input_size = 8
layer_output_sizes = [10, 16, 6]
activation_funcs = [ReLU, ReLU, sigmoid]
```

```
layers = create_layers(network_input_size, layer_output_sizes)

x = np.random.randn(network_input_size)
output=feed_forward(x, layers, activation_funcs)

print("Using ReLU in the hidden layers and sigmoid in the output layer:")
print (output)
```

```
Using ReLU in the hidden layers and sigmoid in the output layer:
[0.05228192 1.         0.59999186 0.93457593 0.99786026 0.99999997]
```

**c)**

In [ ]:
```
activation_funcs_2 = [sigmoid, sigmoid, ReLU]
output_2=feed_forward(x, layers, activation_funcs_2)

print("Using sigmond in the hidden layers and ReLU in the output layer:")
print (output_2)
```

```
Using sigmond in the hidden layers and ReLU in the output layer:
[0.         2.72771082 4.25366266 0.65139649 0.         0.98143355]
```

If you use sigmoid in the hidden layers and ReLU in the output layer, outputs are non negative since ReLU sets negative results to 0.

# Exercise 5: Processing multiple inputs at once

**a)**

In [ ]:
```
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 is the transpose
        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
```

**b)** The function feed_forward_batch:

In [ ]:
```
inputs = np.random.rand(1000, 4)

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

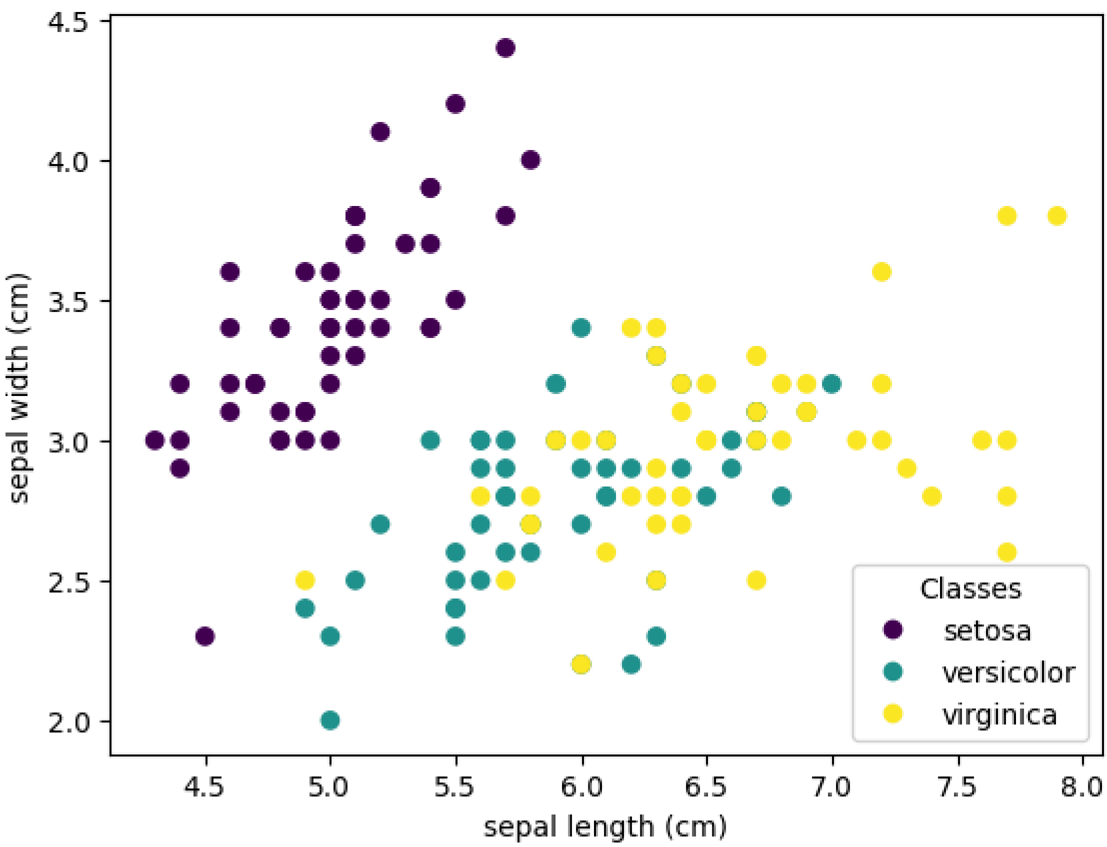
**c)** Creation and evaluation of a neural network with 4 inputs and layers with output sizes 12, 10, 3 and activations ReLU, ReLU, softmax.

In [ ]:
```
network_input_size = 4
layer_output_sizes = [12, 10, 3]
activation_funcs = [ReLU, ReLU, softmax]
layers = create_layers_batch(network_input_size, layer_output_sizes)

x = np.random.randn(network_input_size)
output=feed_forward_batch(inputs, layers, activation_funcs)

print(output)
print(output.shape)
```

```
[[4.04577472e-02 2.35901721e-04 9.59306351e-01]
 [2.45071088e-03 2.87778858e-05 9.97520511e-01]
 [6.62736443e-01 3.36936307e-01 3.27249462e-04]
 ...
 [1.12888793e-01 6.92686503e-04 8.86418520e-01]
 [4.33867835e-03 6.20348500e-06 9.95655118e-01]
 [9.13896494e-01 8.07059531e-03 7.80329107e-02]]
(1000, 3)
```

# Exercise 6 - Predicting on real data

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

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



```
In [ ]: inputs = iris.data

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


        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?

```
In [ ]: print("Input size for the network:", inputs.shape[1])
        print("Output sizes for the network:", targets.shape[1])
```

```
Input size for the network: 4
Output sizes for the network: 3
```

**b)** Creating a network with two hidden layers, the first with sigmoid activation and the last with softmax

```
In [ ]: input_size = inputs.shape[1]
        layer_output_sizes = [8, targets.shape[1]]

        activation_funcs = [sigmoid, softmax]

        layers = create_layers_batch(input_size, layer_output_sizes)
```

**c)** Evaluating the model on the entire iris dataset

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

        print("First prediction:", predictions[0])
        print("Predictions shape", predictions.shape)
        print("Sum of first prediction vector:", np.sum(predictions[0]))
```

```
First prediction: [0.2726041 0.312095  0.4153009]
Predictions shape (150, 3)
Sum of first prediction vector: 1.0
```

**d)** Compute the accuracy

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

```
0.14
```

```
In [ ]: for i in range(5):
            layers = create_layers_batch(input_size, layer_output_sizes)
            predictions = feed_forward_batch(inputs, layers, activation_funcs)
            print(f"Accuracy {i+1}:", accuracy(predictions, targets))
```

```
Accuracy 1: 0.31333333333333335
Accuracy 2: 0.52
Accuracy 3: 0.36666666666666664
Accuracy 4: 0.30666666666666664
Accuracy 5: 0.0
```