# Week 42

This week's exercise used the code snippets from https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/exercisesweek42.html and filled out the missing code.

In [102…
```
import autograd.numpy as np  # We need to use this numpy wrapper to make aut
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 matr
    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
    e_z = np.exp(z - np.max(z))
    return e_z / np.sum(e_z)
```

# Exercise 1

In [102…
```
np.random.seed(2024)

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

```
(2,)
(4, 2)
```

### 1-a)

Given the shape of the input vector $x$, the input shape is $R^{1\times2}$ (a 1x2 vector).

Given the shape of the first layer $W1$, the output of the first layer is $R^{1\times4}$ (4x1 vector).

### 1-b)

In [102…
```
b1 = np.zeros(4) + 0.1
b1 = np.random.randn(4)
```

### 1-c)

In [102…
```
z1 = np.matmul(W1, x) + b1
```

### 1-d)

In [102…
```
a1 = ReLU(z1)
print(a1)
```

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

Test the results

In [102…
```
sol1 = np.array([0.60610368, 4.0076268, 0.0, 0.56469864])

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

```
True
```

## Exercise 2

### 2-a)

The input to the second layer is $R^{4x1}$ (4x1 vector), and the output is $R^{8x1}$ (8x1 vector).

### 2-b)

In [102…
```
W2 = np.random.randn(8,4)
b2 = np.random.randn(8)
```

In [103…
```
z2 = np.matmul(W2,a1) + b2
a2 = ReLU(z2)
print(a2.shape)
```

```
(8,)
```

In [103…
```
print(
    np.allclose(np.exp(len(a2)), 2980.9579870417283)
)  # This should evaluate to True if a2 has the correct shape :)
```

```
True
```

# Exercise 3

## 3-a)

In [103…
```
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
```

In [103…
```
def feed_forward_all_relu(layers, input):
    a = input
    for W, b in layers:
        z = np.matmul(W,a) + b
        a = ReLU(z)
    return a
```

In [103…
```
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)
```

```
[5.36337158 0.        ]
```

## 3-d)

In the *feed_forward_all_relu()* function, we see that the output from one layer is given by $W_i a_i + b_i$, where $a_i$ is the output from the previous layer (/input). The ouput $a_{i+1}$ is then given by the result of the previous equation after going through the activation function. If there were no activation function it would instead be $a_{i+1} = W_i a_i + b_i$. For the next layer this could then be rewritten as $a_{i+2} = W_{i+1} a_{i+1} + bi + 1$, or alternatively $W_{i+1}(W_i a_i + b_i) + bi + 1$. This holds for any number of layers, and is equivilant to a neural network with only one layer.

# Exercise 4

In [103…

```
# For testing
x = np.random.randn(8)
np.random.seed(2024)
```

In [103…

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

In [103…

```
network_input_size = 8
layer_output_sizes = [10, 6, 2]
activation_funcs = [ReLU, ReLU, sigmoid]
layers = create_layers(network_input_size, layer_output_sizes)

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

Out[1037]:

```
array([0.2160035 , 0.00678714])
```

In [103…

```
network_input_size = 8
layer_output_sizes = [8, 4, 2]
activation_funcs = [sigmoid, sigmoid, ReLU]
layers = create_layers(network_input_size, layer_output_sizes)

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

Out[1038]:

```
array([0., 0.])
```

From this single test, it seems the network seems to trend more towards 0.

# Exercise 5

In [103…

```
# For testing
np.random.seed(2024)
```

In [104…

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

        i_size = layer_output_size
    return layers
```

In [104…
```
inputs = np.random.rand(1000, 4) # This needs to be commented out for the te

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

In [104…
```
network_input_size = 8
layer_output_sizes = [8, 4, 2]
activation_funcs = [sigmoid, sigmoid, ReLU]
layers = create_layers_batch(network_input_size, layer_output_sizes)


feed_forward_batch(x, layers, activation_funcs)
```

Out[1042]:
```
array([0., 0.])
```

In [104…
```
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)

predict = feed_forward_batch(inputs, layers, activation_funcs)
print(predict.shape)
```

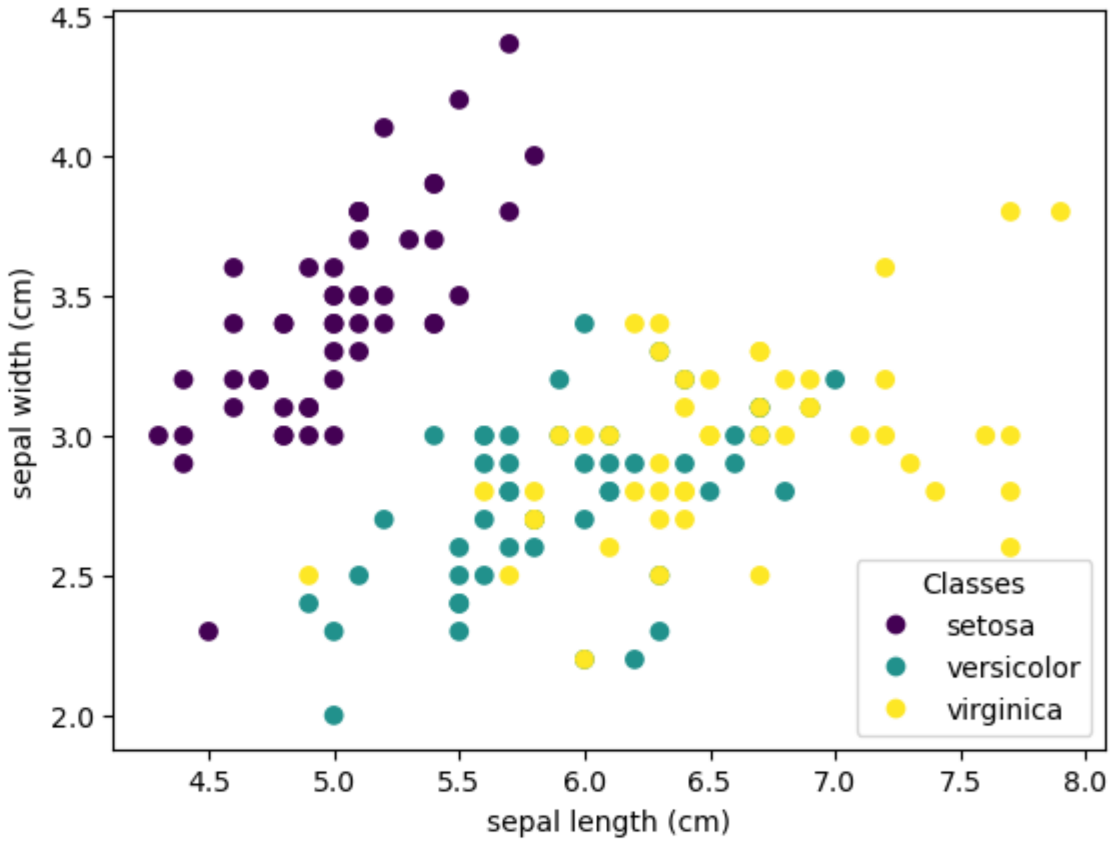
```
(1000, 3)
```

# Exercise 6

In [105…
```
np.random.seed()

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", titl
)
```




In [105…

```
inputs = iris.data

# Since each prediction is a vector with a score for each of the three types
# we need to make each target a vector with a 1 for the correct flower and a
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)
```

In [105…

```
print(inputs.shape)
```

```
(150, 4)
```

The inputs should be 4, and the output should be 3

In [109…

```
network_input_size = 4
layer_output_sizes = [8, 3]
# layer_output_sizes = [4, 3]
activation_funcs = [sigmoid, softmax]
# activation_funcs = [sigmoid, sigmoid]
layers = create_layers_batch(network_input_size, layer_output_sizes)

predictions = feed_forward_batch(inputs, layers, activation_funcs)
print(predict.shape)
print(targets.shape)
print("Accuracy: " + str(accuracy(predictions, targets)))
```

```
(1000, 3)
(150, 3)
Accuracy: 0.25333333333333335
```

Changing the activation functions to both be sigmoid makes the accuracy less varied when running multiple times.

# Exercise 7

In [104…

```
# Setup (Same as exercise 6)
iris = datasets.load_iris()
inputs = iris.data
targets = np.zeros((len(iris.data), 3))
for i, t in enumerate(iris.target):
    targets[i, t] = 1
```

In [105…

```
from autograd import grad


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