

---

**General hints:**

- Your code should work with *Python 3.8*.
- Add comments to your code, to help us understand your solution.
- We suggest to adhere to the [PEP8](#) style guide and use line lengths of up to 120.
- `for` loops can be slow in Python, use vectorized `numpy` operations wherever possible (see assignment 1 for an example).

---

**How to run the exercise and tests:**

- See the setup instructions downloadable from our website for installation details.
- We always assume you run commands in the *root folder* of the exercise.
- If you're using miniconda, don't forget to activate your environment with `conda activate mycvenv`
- Install the required packages with `pip install -r requirements.txt`
- Python files in the *root folder* of the repository contain the scripts to run the code.
- Some exercises contain unittests in folder `tests/` to check your solution. Run `python -m pytest .` to run all tests.
- To check your solution for the correct code style, run `pycodestyle -max-line-length 120 .`

---

In this course, you will learn about Computer Vision. In this first exercise you will set do some small exercises to brush up your *linear algebra* and to get familiar with *python* and *numpy*.

At this point you should have worked through the `setup.pdf` and have a working python 3.8 installation ready. You should know how to navigate and run commands in the command prompt.

**1. Pen and Paper tasks**

Now you'll brush up your linear algebra a little by doing eigendecomposition by hand. Later we will see how we can do the same in python.

If you'd like to review Linear Algebra, we recommend [Khan Academy](#). [The Matrix Cookbook](#) is another useful resource.

- 1) Calculate the eigendecomposition of the following matrix:

$$A = \begin{bmatrix} 9 & 1 \\ 8 & 7 \end{bmatrix}$$

- 2) Use the eigendecomposition of  $A$  above to show how you can efficiently compute  $A^{10}$  (you don't have to show the final value of the matrix).

- 3) You are given the following matrix:

$$A = \begin{bmatrix} 4 & 8 & 2 \\ 8 & 41 & 24 \\ 2 & 24 & 21 \end{bmatrix}$$

$$\det(A) = 400$$

One of the eigen values is 1. Find the other two. **Hint:** You don't have to calculate the eigenvalues from scratch. Use the properties of eigenvalues.

- 4) You are given the following matrix:

$$A = \begin{bmatrix} 100 & 100 & 100 \\ 99 & 99 & 102 \\ 98 & 98 & 104 \end{bmatrix}$$

$$\det(A) = 0$$

Find the eigenvalues of  $A$ .

## 2. Code Warmup

To solve code exercises, you have to fill in code between the *START TODO* and *END TODO* markers in the code. Before you run any code, make sure you have the correct conda environment activated, and don't forget to install the required packages with `pip install -r requirements.txt`.

**Todo:** In the file `lib/example_file.py`, complete `example_function` by adding

```
return input_variable * 2
```

Execute the command `python example_script.py` to see the function in action.

Run `python -m tests.test_example` to test if the function is implemented correctly.

## 3. Getting to know numpy

### 1) Numpy tensors

You will now play around with some basics of tensor manipulation in *numpy*. The basic object in numpy is an homogeneous multidimensional array. Numpy's array class is called *ndarray*. Here is a quickstart tutorial: <https://numpy.org/devdocs/user/quickstart.html>

**Todo:** Run the script `run_numpy_arrays.py`. We will walk you through it's code and output during this exercise.

Let's create two matrices and check their properties.

```
A = np.array(np.arange(4))
B = np.array([-1, 3])
print(f"A (shape: {A.shape}, type: {type(A)}) = {A}")
print(f"B (shape: {B.shape}, type: {type(B)}) = {B}")
```

#### Output:

```
A (shape: (4,), type: <class 'numpy.ndarray'>) = [0 1 2 3]
B (shape: (2,), type: <class 'numpy.ndarray'>) = [-1  3]
```

First, 2 arrays (also called tensors in the context of deep learning) are created. Each numpy tensor has an attribute `numpy.ndarray.shape` which describes the dimensions of the defined tensor. Type, shape and content of the tensors are the first output of the script. Please note how we are using **f-strings** to output variables.

In order to perform matrix multiplication and addition in numpy there are two methods: `numpy.matmul` and `numpy.add`. Please read their respective documentation in numpy before proceeding.

Next, we try to multiply the two tensors with `matmul`.

```
np.matmul(A, B)
```

#### Output:

```
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0,
with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 2 is different from 4)
```

We get a *ValueError* due to the shape mismatch between the two numpy arrays we want to multiply/add. In order to deal with different array shapes during arithmetic operations, we can either **reshape** the arrays or **broadcast** the smaller array across the larger one such that they have compatible shapes.

```
C = A.reshape([2, 2])
print(f"C shape: {C.shape}, content:\n{C}")
```

#### Output:

```
C shape: (2, 2), content:
[[0 1]
 [2 3]]
```

Now the matrix multiplication  $CB$  works out.

```
matmul_result = np.matmul(C, B)
print(matmul_result)
```

**Output:**

```
[3 7]
```

When adding the  $C$  with shape  $(2, 2)$  and  $B$  with shape  $(2, )$ ,  $B$  will be automatically broadcast to match the shape of  $C$ .

```
print(np.add(C, B))
```

**Output:**

```
[[-1  4]
 [ 1  6]]
```

The star operator  $*$  will do an element-wise multiplication between the  $C$  and  $B$ . Again,  $B$  will be broadcasted to fit.

```
print(C * B)
```

**Output:**

```
[[ 0  3]
 [-2  9]]
```

The function `np.diag` can transform the vector  $B$  shaped  $(2, )$  into a diagonal matrix of shape  $(2, 2)$ .

```
print(np.diag(B))
```

**Output:**

```
[[-1  0]
 [ 0  3]]
```

For transposing a ndarray use `numpy.transpose` or the method `numpy.ndarray.T`.

```
print(np.transpose(C))
```

**Output:**

```
[[0 2]
 [1 3]]
```

Tensor operations are a central part of the exercises and deep learning in general, so play around with the script to get familiar with them. You could also just start python with the command `python` and play around in there.

## 2) Remember that for loops can be slow

Use vectorized numpy expressions instead of manual loops wherever possible. The following is an example for computing the sum  $\sum_{i=0}^{N-1} i^2$  with  $N = 1000000$ :

This is **wrong** (Takes about *200ms*).

```
total = 0
for i in range(1000000):
    total += i ** 2
```

This is **correct** (Takes about 8ms):

```
import numpy as np
numbers = np.arange(1000000, dtype=np.int64)
total = (numbers ** 2).sum()
```

**Note:** The `np.int64` datatype means we use integers that are large enough to store the result. The square and sum operations are vectorized and run in fast C code internally.

### 3) Eigendecomposition

Using `numpy.linalg` you can also perform many linear algebra functionalities. Given a square and symmetric matrix  $A$ , the eigendecomposition  $A = Q\Lambda Q^T$  with  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$  can be done using `numpy.linalg.eig`.

**Todo:** Run the script `run_eigen.py` to see the eigendecomposition in action for  $A = \begin{bmatrix} 7 & -\sqrt{3} \\ -\sqrt{3} & 5 \end{bmatrix}$

Do not worry about the `NotImplementedError`, you will fix that now.

**Todo:** In file `lib/eigendecomp.py`, complete the function `get_matrix_from_eigdec` to return the square matrix  $A$ , given its eigenvalues  $\lambda_1, \dots, \lambda_n$  and eigenvectors  $Q$  as an input. **Note:** We are using `type hints` to make the code more readable and give you hints about the input and output.

**Todo:** Complete the `get_euclidean_norm` and `get_dot_product` functions. These take vectors as input and are used to show that the columns of  $Q$  are orthonormal, i.e. the columns are of unit length and the columns are orthogonal (their dot product is 0).

**Todo:** Complete the `get_inverse` function by using that  $A^{-1} = Q\Lambda^{-1}Q^T$  with  $\Lambda^{-1} = \text{diag}(\lambda_1^{-1}, \dots, \lambda_n^{-1})$  is the inverse of  $A$ . Do **not** use `numpy.linalg.inv`. You can invert the diagonal matrix  $\Lambda$  without it.

### 4) Vector Norms

The length of a vector is not a single number but can be defined in different ways. These vector norms share common properties but also have different characteristics. In numpy you can use the `numpy.linalg.norm` function to compute the  $L_p$  norm of a numpy array, where  $p \geq 1$ .

More formally, the  $L_p$  norm of a vector  $x$  is defined as:  $\|x\|_p = (\sum_{i=1}^n |x_i|^p)^{1/p}$

For  $p = 1$  we get the Manhattan norm, for  $p = 2$  we get the Euclidian norm and for  $p = \infty$  we approximate the maximum norm:  $\|x\|_\infty = \max_i |x_i|$ .

Now to plot the norms, we create a 2-dimensional grid and color the norm values as a heat map. To create the grid, we use `meshgrid` which can transform two 1-dimensional vectors into a grid representation. For example:

```
N = 3
lin_x = np.linspace(-1, 1, N) # e.g. for N=3: [-1, 0, 1] with shape (3,)
lin_y = np.linspace(-1, 1, N) # same
X, Y = np.meshgrid(lin_x, lin_y)
print(f"X (shape {X.shape}):\n{X}\n")
print(f"Y (shape {Y.shape}):\n{Y}\n")
```

**Output:**

```
X (shape (3, 3)):
[[-1.  0.  1.]
```

```
[-1.  0.  1.]  
[-1.  0.  1.]]
```

```
Y (shape (3, 3)):  
[[-1. -1. -1.]  
 [ 0.  0.  0.]  
 [ 1.  1.  1.]]
```

This way, you can get the 2D-coordinates of the grid at  $i, j$  by accessing  $X_{i,j}$  and  $Y_{i,j}$

**Todo:** in file `lib/norms.py`, complete the function `get_norm` to compute the norm of each 2D vector composed by the  $i, j$ -th elements of the matrices  $X$  and  $Y$ . You will first need to stack the two matrices together to form a  $(N, N, 2)$  tensor (using `np.stack`) and then compute the  $L_p$ -norm (using `np.linalg.norm`). This way, you get the norm results for each point in the grid. Run the file `plot_norms.py` to see a plot of the 2-norm.

We have added an argument to the script with the `argparse` package. Run the command `python plot_norms.py -p 1` to see a plot of the 1-norm.

**This assignment will be discussed on 27.10. 12:00-13:00 CET.**

Special thanks to the team of the course Foundations of Deep Learning, Machine Learning Lab, University of Freiburg for allowing us to use this exercise in our course.