Deep Learning
1st Assignment

To get access to this week's code use the following link: https://classroom.github.com/a/3B19tVke

General constraints for submissions: Please adhere to these rules to make our and your life easier! We will deduct points if you fail to do so.

- Your code should work with *Python 3.8*.
- You should only fill out the *TODO-gaps* and not change anything else in the code.
- Add comments to your code, to help us understand your solution.
- Your code should adhere to the PEP8 style guide. We allow line lengths of up to 120.
- While working on the exercise, push all commits to the `dev` branch (details in assignment 1). Only push your final results to the `master` branch, where they will be automatically tested in the cloud. If you push to `master` more than 3 times per exercise, we will deduct points.
- All provided unit tests have to pass: In your *GitHub* repository navigate to *Actions* → your last commit → Autograding → education/autograding to see which tests have passed. The points in autograding only show the number of tests passed and have nothing to do with the points you get for the exercise.
- `for` loops can be slow in Python, use vectorized `numpy` operations wherever possible (see assignment 1 for an example).
- Submit *a single PDF* named `submission.pdf`. Include your matriculation numbers on the top of the sheet. Add the answers and solution paths to all non-coding questions in the exercise. Do not leave answers to any questions as comments in the code. You can use Latex with the student template (provided in exercise 1 / ILIAS) or do it by hand.
- Please help us to improve the exercises by filling out and submitting the `feedback.md` file.
- We do not tolerate plagiarism. If you copy from other teams or the internet, you will get 0 points. Further action will be taken against repeat offenders!
- Passing the exercises ($\geq 50\%$ in total) is a requirement for passing the course.

How to run the exercise and tests

- See the `setup.pdf` in exercise 1 / ILIAS for installation details.
- We always assume you run commands in the *root folder* of the exercise repository.
- If you use miniconda, do not forget to activate your environment with `conda activate mydlenv`
- 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.
- Python files in the `tests/` folder of the repository contain the tests that will be used to check your solution.
- Test everything at once with `python -m pytest`
- Run a single test with `python -m tests.test_something` (replace `something` with the test's name).
- To check your solution for the correct code style, run `pycodestyle --max-line-length 120 .`
- The scripts `runtests.sh` (Linux/Mac) or `runtests.bat` (Windows) can be used to run all the tests described above. If you are on Linux, you need execution rights to run `runtests.sh`.

In this course, you will learn about the *foundations* of deep learning. Instead of going in-depth into any particular topic, the lectures will cover a wide variety of deep learning methods and their underlying principles. Similarly, these exercises will *not* train you to be a specialist deep learning practitioner, but rather aim to give you a deeper understanding of the canonical methods, by implementing them yourself and by applying them to some classical benchmark problems.

In this first exercise you will set up teams and learn about git and the workflow for future assignments. You will also do some small exercises to brush up your *linear algebra* and *probability theory*, 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 and git installation ready. You should know how to navigate and run commands in the command prompt.

1. [2 points] **Form teams of 3 students**

Exercises have to be completed in teams of up to 3 students. You can use the "Let's Team Up" thread on the ILIAS forum to find team members.

When you have found your partner(s), open the following link `https://classroom.github.com/a/3B19tVke`, create a group (your team name must start with `dl2023-`, e.g. `dl2023-my_team`) and have both your colleagues join that group.

This will allow you to clone the template repository in which you can add your solutions to this exercise sheet.

Note: Make sure you and your team-mates are happy with each other. We will only allow changing your groups mid semester if you have a very good reason to do so. **Note:** If you have not found a team yet and want to take a look at the exercise sheet, we have uploaded the code to the ILIAS forum in the same folder as you found this pdf. However, the submission needs to be made through GitHub as part of your team.

2. [1 point] **Upload your names on GitHub**

Add a file called `members.txt` to your repository.
The file should contain the names of all members in the following way:

```
name 1; mail address 1; ILIAS username 1
name 2; mail address 2; ILIAS username 2
name 3; mail address 3; ILIAS username 3
```

If you have fewer members, add fewer lines.

We make use of GitHub Classroom's testing functionality. Essentially, for most exercise sheets we will require you to pass unit tests which are automatically evaluated whenever you push to GitHub. For example, for this exercise we run a test that expects the `members.txt` file to be present and checks if it is filled out correctly (valid emails and ILIAS usernames).

Testing costs build time on our server. It is only enabled when pushing to the `master` branch of your repository. You are only allowed to push to `master` up to 3 times – we will deduct points if you push more than that. Instead, run the tests locally (see "How to run the exercise and tests" above), push on the `dev` branch and only push your final solution to the `master` branch.

Here is how to create the `dev` branch:

- `git pull` to make sure you are up to date.
- `git switch -c dev` to create the `dev` branch and switch to it.
- `git push --set-upstream origin dev` to connect your local dev branch with the repository.

Here is how to switch to the `dev` branch if it has already been created:

- `git pull` to make sure you are up to date.
- `git branch -a` to list all branches, you should see the `dev` branch.
- `git fetch -p` is only needed if you do **not** see the `dev` branch.
- `git switch dev` to work on the `dev` branch.

Now that you are working on the `dev` branch, upload the file to your repository on *github*. To do this, run the following commands:

- `git pull` to make sure you are up to date.
- `git status` to see what has changed locally.
- `git add .` to add all changes (be careful to not upload the wrong files. If you want to upload only some files, either change the parameters of the `git add` command or change the `.gitignore` file.)
- `git status` again to see what has been added with the last command.
- `git commit -m "update members.txt"` to commit.
- `git push` to upload your changes.

Now, merge the `dev` branch to the `master` branch and push to see the auto-testing in action. This push to `master` does not count towards the maximum 3 pushes you are allowed to do per exercise.

- `git pull` to make sure you are up to date.

- `git switch master` to switch from the `dev` to the `master` branch.
- `git merge dev` to merge `dev` into `master`. If you get merge conflicts, see e.g. this link on how to resolve them. Basically you will have to change the files by hand and then add and commit them.
- `git push` to upload the merge.

After uploading, in your repository on github under *Actions* → your last commit → Autograding → education/autograding , you can find out whether your `members.txt` file passed the `test_names` test.

Note: You can have as many branches as you want. It *can* make sense to create branches for each person working on the repository and merging them to `dev`, then merging to `master` at the end. However, you can also just all work on `dev` at the same time and merge as needed.

See this link for an overview of how git branches are used in a professional environment.

3. [2 points] **Pen and Paper task**

Provide your solution as a PDF file. You can use the template tex file we provided you for this.

Probability Theory: Bertrand's Box Paradox

Now you will test your knowledge of elementary probability theory. Probability theory plays a central role in machine learning in general, and specifically in the later lecture on uncertainty quantification in deep neural networks.[1]

Consider a box containing the following 3 cards:

- one with two black sides
- one with two white sides
- one with a black and a white side

From the three cards, one is drawn at random and put on the table. You can only see the side facing up.

Todo: Answer the following questions in your `submission.pdf` (to be solved mathematically, providing intermediate steps and explanations):

1. What are the probabilities that the card on the table shows a black side? What are the probabilities it shows a white side?
2. If we draw a card and it shows black, compute the probability that the other side of the card is also black.
3. Find the probability that the other side of the card is black if the card shows a white side.

4. [1 point] **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` . In general, you can use any imported methods from packages in the environment, unless otherwise specified in the todo-block.

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.

5. **Getting to know numpy**

[1] We provide a refresher course on ILIAS introducing the core concepts required for that specific lecture.

1) Numpy tensors

You will now play around with some basics of tensor manipulation in *numpy*. The basic object in numpy is a 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 its 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. 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 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 broadcast 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 an interactive python session 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. We will **deduct points** if your code is too slow. The following are examples for computing the sum of one million random numbers between zero and one:

This is **wrong** (Takes about *400ms*).

```
import numpy as np
numbers = np.random.random(1000000)
total = 0
for number in numbers:
    total += number ** 2
```

This is the **correct** way to circumvent the necessity of a loop (Takes about *8ms*):

```
import numpy as np
numbers = np.random.random(1000000)
total = (numbers ** 2).sum()
```

Note: The square and sum operations are vectorized and run in fast C code internally.

3) [4 points] **Eigendecomposition**

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

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 $\Lambda = diag(\lambda_1, \ldots, \lambda_n)$ 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 symmetric matrix A, given its eigenvalues $\lambda_1, \ldots, \lambda_n$ 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.

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

Todo: Complete the `get_euclidean_norm` function without using `numpy.linalg.norm` to show that you know how to avoid using a manual loop in favor of a vectorized numpy expression.

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

Todo: Complete the `get_dot_product` function.

The two functions mentioned above 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 are pairwise orthogonal (their dot product is 0).

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

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

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

6. Distributions and the Central Limit Theorem

1) [1 point] **Central Limit Theorem**

The central limit theorem states that for i.i.d. random samples $\{X_i\}$ from an (almost) arbitrary distribution with given mean μ and variance σ^2, the mean $\frac{1}{n}\sum_{i=1}^{n} X_i$ follows approximately a normal distribution. More precisely, it reads

$$\frac{1}{n}\sum_{i=1}^{n} X_i \xrightarrow{n\to\infty} \mathcal{N}(\mu, \frac{\sigma^2}{n})\,.$$

Instead of proving this theorem, we are going to simulate the theorem experimentally. To do so, we are going to draw $n = 1, 16, 64, 1024$ samples from the distributions below and calculate the sample mean. This mean should be normally distributed. To evaluate if this is true, we are going to repeat this process 1024 times (for each n and each distribution separately).

- the exponential distribution $p(X) = \lambda \mathrm{e}^{-\lambda X}$ with $\lambda = 1$
- the Gaussian/normal distribution with $\mu = 1, \sigma = 1$

Todo: Complete function `plot_clt` in file `lib/distributions.py` and run file `plot_clt.py`. Calculate for each (n, distribution) pair the repeated samples and their means. We took care of the plotting code and included the corresponding normal distribution (probability density function) in the plot for you so you can compare the results. Briefly discuss how the generated plots relate to the theorem.

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

2) [1 point] **Uniform Distributions**

Now assume that you can only sample from uniform distributions. Implement functions to sample from an approximated standard normal distribution and an approximated normal distribution. Plot the distributions in comparison to the numpy implementation.

Todo: Complete functions `std_normal` and `normal` in file `lib/distributions.py` and run file `plot_normal.py`

Hint: Use the Central Limit Theorem: Draw $M * N$ samples from a uniform distribution and calculate the M means of those samples. Those means then approximate a norm distribution.

The variance of a uniform distribution $U(a, b)$ is $\sigma_u^2 = \frac{1}{12}(b-a)^2$
So for a uniform distribution $U(-b, b)$ the variance is $\sigma_u^2 = \frac{1}{12}(2b)^2 = \frac{1}{3}b^2$
The Central Limit Theorem states that the means of a number of samples from this uniform distribution will follow a normal distribution with the same mean as the uniform distribution and variance $\sigma_n^2 = \frac{\sigma_u^2}{n_u}$ where n_u is the number of samples.
Given that we want to sample from a standard normal distribution we know that $\sigma_n = 1$.

Solve for b to know from which uniform distribution you need to sample in the task above.

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

7. Bayesian Linear Regression (BLR)

To begin with, we want to familiarize you a bit more with Bayesian Linear Regression. We will need BLR later for the implementation of the DNGO model and it is also a nice starting point to get familiar with sampling from distributions of models that can explain our observations. We already provided you with the class `BLR` which implements Bayesian Linear Regression. As we have seen in the lecture BLR assumes a prior distribution over model weights which we will assume to be Gaussian with mean μ_p and covariance matrix Σ_p.
When observing training data, BLR will compute a posterior distribution over regression weights defined by μ_{post} and Σ_{post}, which you can access and use for your implementation. To compute this posterior distribution, `linreg_bayes` has to be called. `BLR` also provides a method to compute the posterior predictive distribution which can be called via `posterior_predictive`. It is also important to note here, that by setting `bias = True` BLR can approximate functions that do not pass through the origin.
Note: In case you need more details on BLR, you can have a look at the GPML book. We also provide material on BLR and Properties of Gaussians in the background material.

1) [2 points] **Plot Samples from Prior/Posterior** As a start we want to figure out how we can generate different linear models using BLR. We will sample from the prior and posterior distribution that we get from fitting the Bayesian Linear Regression (`linreg_bayes`) and plot the different linear models corresponding to each sampled weight.

   Todo: Complete the function `plot_bayes_linear_regression` in `lib/plot.py` to plot the samples from the prior distribution and additionally (on top) the samples from the posterior to see the differences. Make sure to plot the models sampled from the prior/posterior in two different colors. Also include the prior mean (μ_p) and posterior mean (μ_{post}) in the plot. Finally, use varying alpha (opacity) values to indicate the relative likelihood of models ($\sim$ probability density), i.e. use a higher transparancy (lower alpha $\sim$ lighter color) for less likely models.
   To see the plot you can run `run_lin_regression.py`, however it is meant to be run together with the contour plot you will implement next.

   Note: We assume the biases to be 0 in this part

2) [2 points] **Plot Contour & Answer Question** Now we perform BLR for a 2d dataset and generate contour plots of the pdf of the prior and posterior distributions.

   Todo: Complete `plot_contour` in `lib/plot.py`. Your task is to use the data and compute the Gaussian posterior with `linreg_bayes`. Furthermore, you should instantiate the multivariate Gaussian distributions for the prior and the posterior, using `scipy.stats.multivariate_normal`.

   After completing the functions, run `run_lin_regression.py` to see the plots.

   Answer in pdf: How do the prior and posterior distributions differ? Why?

   Hint: You should explain the difference between the prior and posterior distributions, and *not* the difference between the 1D and 2D plot.

8. [1 bonus point] Code Style

On every exercise sheet, we will also make use of `pycodestyle` to adhere to a common python standard. Your code will be automatically evaluated on submission (on push to master). Run `pycodestyle --max-line-length=120 .` to check your code locally.

9. [1 bonus point] Feedback

Todo: Please give us feedback by filling out the `feedback.md` file.

- Major Problems?
- Helpful?
- Duration (hours)? For this, please follow the instructions in the `feedback.md` file.
- Other feedback?

This assignment is due on 31.10.2023 23:59 CEST. Submit your solution for the tasks by uploading (`git push`) the PDF, txt file(s) and your code to your group's repository. The PDF has to include the name of the submitter(s). Teams of at most 3 students are allowed.