To get access to this week's code use the following link: https://classroom.github.com/a/t-XEEPKp

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 `main` branch, where they will be automatically tested in the cloud. If you push to `main` 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`.

1. [3½ points] **Pen and Paper MLP**

Consider a binary classification problem, where each data point belongs to one of two classes 0 or 1.

We have an MLP that, given some input data, predicts the probability of that input data belonging to class 1.

Todo: Given the following inputs and labels as well as the weights, biases and activation functions of a 2-layer MLP, calculate by hand the forward pass and the Binary Cross-Entropy-Loss (Cross-Entropy Error) for each datapoint and then average over all datapoints. (Use the natural logarithm unless explicitly stated otherwise.)

Inputs and labels: $N = 2$ data points, $x_1 = (1, 2, 3)^\top$, $x_2 = (3, 4, 5)^\top$, $y_1 = 0$, $y_2 = 1$

Parameters of the MLP:

$W^{(1)} = \begin{pmatrix} -2 & 1 \\ 2 & 0 \\ -3 & 1 \end{pmatrix}$, $b^{(1)} = (3, 0)^\top$, $g^{(1)}$: ReLU

$W^{(2)} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$, $b^{(2)} = -3$, $g^{(2)}$: Logistic sigmoid activation function

2. Your first implementation

Exercises 3 and 4 focus on implementing and running a small feedforward neural network. We recommend looking at Chapter 6 in the "Deep Learning" book.

We supply you with starter code providing a modular structure and basic functionality, and you have to **fill in the TODO-gaps.** It might initially look like a lot of unnecessary code, but it keeps the network extensible. In the following exercises you can reuse what you've done here. Most common neural network libraries (*pytorch*, *tensorflow/keras*, ...) are similarly structured, hence they will be easy to use once you've finished this exercise sheet. As we will be using *pytorch* towards the end of the lecture, our API resembles the API of the pytorch framework.

We will implement two different Loss functions and play a bit with the value of the different hyperparameters to see how performance changes as a function of those. In particular, we will focus on

- Nonlinear activation functions
- Multi-Layer Perceptrons (MLP) a.k.a. Feed-forward neural networks
- Quadratic Loss Function
- Cross Entropy Loss Function

Note that we'll implement all of these operations to operate on batches of data.

Network: File `lib/network_base.py` contains the two main classes to structure your network.

Parameter is used to represent trainable variables in the network, e.g., a layer's weights w. The weights themselves are stored as a *numpy array* in the parameter's *data* attribute. The associated parameter gradient (e .g. $\frac{\partial L}{\partial w}$) can be stored in the *grad* attribute.

Module is the base class for all parts of the network (activations, layers, ...) and even the network itself. They all have to implement the `forward` and `backward` methods. When training the network, activations will flow *forward* and gradients will flow *backward* through the *network graph and its sub-modules.*

Additionally, **module** provides functionality to check the correctness of implementation by approximating *backward* with finite difference approximations of *forward.*

Note: All modules operate on batches of samples. E.g. the shape of the input of `Linear.forward` is `(batch_size, feature_shape)`. In this exercise we will not yet make use of gradients and the backwards method. In the next exercise we will see how one can update the weights of a neural network using back-propagation.

1) [1 point] **Todo:** Implement your first network module: The sigmoid activation function (class `Sigmoid` in file `lib/activations.py`). Run `python -m tests.test_sigmoid` to test your implementation.

   Verify your sigmoid function by plotting (run the file `plot_sigmoid.py`).

2) [1 point] **Todo:** Implement the ReLU module (class `ReLU` in file `lib/activations.py`). Run `python -m tests.test_relu` to test your implementation.

3) [½ point] **Todo:** Plot ReLU for verification (fill the gaps in file `lib/plot.py` and run file `plot_relu.py`). Run `python -m tests.test_plot_relu` to test your implementation.

4) [1 point] **Todo:** Implement the Softmax module (class `Softmax` in file `lib/activations.py`).

   Hint: For numerical stability, subtract the maximum value of the input from all inputs. This will not change the solution, but make the calculation numerically stable.

   Run `python -m tests.test_softmax` to test your implementation.

5) [2 points] **Todo:** Implement a linear (in other frameworks also called dense or fully connected) network layer (class `Linear` in file `lib/network.py`). Here you also have to use the Parameter class.

   Hint: You need to use the matrix formulation of the linear layer which is $Z = XW + b$.[1] Broadcasting in numpy will automatically expand B such that the addition of the matrix XW and the vector b works out.

   Run `python -m tests.test_linear` to test your implementation.

6) [1 point] **Todo:** Implement the generalization of the Cross-Entropy Error function (class `CrossEntropyLoss` in file `lib/losses.py`).

   Note: Generalized Cross-Entropy loss means that we have transformed the binary classification problem into a multi-class classification problem with 2 classes. Because of this, we will have 2 outputs (1 output per class).

   Run `python -m tests.test_crossentropy` to test your implementation.

7) [1 point] **Todo:** Implement the `Sequential` class in file `lib/network.py`. Run `python -m tests.test_sequential` to test your implementation.

3. Experiments

We will now implement a simple multilayer perceptron whose output approximates the output of the XOR operation. At this point we are not concerned to generalize; we only care to learn a function that can classify each pair in $X = \{[0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T\}$ to one of the classes in $Y = \{0,1\}$. Note that our input X is a matrix with dimensions $N \times D$ for N examples and D data dimensions.

The goal is to fit a **logistic regression** model to the X data.

1) [½ point] **Question:** What is the best accuracy you can achieve in practice using Logistic Regression?

2) [½ point] **Todo:** Implement the function `logistic_regression` in file `lib/log_reg.py` and run the file `run_log_reg.py`.

   Hint: You can use sklearn.linear_model.LogisticRegression with the LBFGS solver and L2 regularization.

   Run `python -m tests.test_log_reg` to test your implementation.

3) [2½ points] Now we will define a 2-layer **sequential model** (Linear, ReLU, Linear) with 2 hidden units. 2-layer **sequential model** means it has an input layer, a hidden layer and an output layer. The input layer is not counted. Having 2 hidden units in a 2-layer **sequential model** means it has 2 units in its hidden layer. Our model output will therefore be computed as

[1] In this formula, the batch dimension is on the first axis, i.e. rows of X correspond to data points. This is slightly different compared to e.g. PyTorch. In PyTorch the output is calculated as $Z = XW^T + b$. In literature (and in the lecture slides) you will also come across the formula $Z = W^T X + b$, where data points correspond to columns of X.

$f(x) = \max\{0, XW_1 + b_1\}W_2 + b_2$

and we will be using
$W_1 = \begin{bmatrix} 3.21 & -2.34 \\ 3.21 & -2.34 \end{bmatrix}$, $b_1 = \begin{bmatrix} -3.21 & 2.34 \end{bmatrix}$, $W_2 = \begin{bmatrix} 3.19 & -2.68 \\ 4.64 & -3.44 \end{bmatrix}$, $b_2 = \begin{bmatrix} -4.08 & 4.42 \end{bmatrix}$

These parameters are actually the minimizers of some cost, e.g. one of the cost functions shown above. In the next exercise, you will learn how backpropagation is used to find these parameters using gradient descent.

For this part you have to:

- instantiate the sequential model.
- set the parameters of the sequential model to the corresponding minimizer's values as shown above.
- propagate the input examples through the network to get the output.

Note: To handle categorical data, the `one_hot_encoding` utility function is provided in file `lib/utilities.py`. The function is used to convert the input labels from class numbers to one-hot encodings. You will be required to use it now.

Todo: Fill in the gaps for function `run_model_on_xor` in file `lib/models.py`. Run `python -m tests.test_run_model` to test your implementation.

Todo: Then, complete function `create_2unit_net` in file `lib/models.py`.
Run `python -m tests.test_2unit_model` to test your implementation.

Todo: Run the file `run_2unit_model.py` to see the outputs.

4) [1½ points] **Todo:** Instantiate a **new 2-layer sequential model** with 3 hidden units this time. Assign the correct values to the model parameters such that the output of the neural network remains the same as in the model with 2 hidden units.

Todo: Fill in the gaps in function `create_3unit_net` in file `lib/models.py`.

Run `python -m tests.test_3unit_model` and `python -m tests.test_2unit_3unit_model_output` to test your implementation.

Todo: Run the file `run_3unit_models.py` to see the outputs.

4. [1½ bonus points] Plot representation space

After transforming the input data into the hidden representation space, generate two plots showing the dataset in the input and representation spaces, respectively (the same as the one in Figure 6.1, Chapter 6, Deep Learning Book). In order for the plots to be consistent, you have to change the weights of the first Linear layer as follows:

$W_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $b_1 = \begin{bmatrix} 0 & -1 \end{bmatrix}$,

Hint: To extract the hidden representation from a 2-layer MLP, propagate the input through the first Linear layer and the ReLU activation function of the model.

Todo: Fill in the gaps in the `extract_hidden` function in file `lib/model_utilities.py` and run the file `run_bonus.py`. Make sure that you have solved the exercise part above where you filled in the gaps in function `create_2unit_net`, file `lib/models.py`.

Run `python -m tests.test_extract_hidden` to test your implementation.

5. [1 bonus point] **Code Style**

On each exercise sheet, we will also be using `pycodestyle` to adhere to a common Python standard. `pycodestyle` checks your Python code against some common style conventions in PEP 8 and reports any deviations with specific error codes.

Your code will be automatically evaluated on submission (on push to `main`). You can run `pycodestyle --max-line-length=120 .` to manually evaluate your code before submission.

One bonus point will be awarded if there are no code style errors detected in your code by the `pycodestyle --max-line-length=120 .` command.

6. [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 06.11.2024 23:59 CET. 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.