To get access to this week's code use the following link: https://classroom.github.com/a/FXp8bW-w

---

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`.

This exercise focuses on optimization, we will:

- Calculate update steps of SGD with momentum by hand
- Implement the Adam optimizer
- Compare Adam, SGD, SGD with momentum
- Implement and analyze some learning rate schedules
- Analyze the effect of ill-conditioning and how it can be mitigated using momentum or preconditioning

1. Pen and Paper tasks

1) [2 points] Perform two update steps of SGD with momentum to the weights $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ of the linear function $\hat{\mathbf{y}} = \mathbf{w}^\top X$ optimizing the mean squared error (MSE). Thus, the loss for one sample is given by $(\hat{y} - y)^2$ and according to slide 43 in the optimization lecture, the per-sample losses are summed and divided by the batch size.

Given:

the initial state $\mathbf{w} = \begin{bmatrix}1\\0\end{bmatrix}$, velocity $\mathbf{v} = \begin{bmatrix}0 & 0\end{bmatrix}$, data $X = \left[\begin{bmatrix}2\\-1\end{bmatrix}\begin{bmatrix}-1\\3\end{bmatrix}\right]$, target $\mathbf{y} = \begin{bmatrix}3 & 1\end{bmatrix}$, learning rate $\alpha = 0.2$, momentum strength $\beta = 0.8$ and batch size $B = 2$.

2) [2 points] Prove that bias corrected first order moment s (i.e., $\hat{s}$) is equal to the gradient for every step in Adam when that gradient remains constant.

Hint: This can be done using mathematical induction.

2. Coding Tasks

In this coding exercise, you will implement an Adam optimizer and compare it with SGD and SGD with momentum. You will also implement two learning rate schedulers - one which uses piecewise constants and another which uses cosine annealing.

Please note that we have extended the Module class in `lib/network_base.py` with a state (`self.training = True`) to determine whether we're training or evaluating the module. We also added two methods to toggle this state - `train()` and `eval()`. You will see these methods being used in the `train` function in `lib/training.py`.

1) [2 points] **Todo:** Implement the step function of the adam optimizer (class `Adam` in file `lib/optimizers.py`) **Hint:** We provide an implementation of SGD in the same file.

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

2) [2 points] In the `lib/training.py` file, train three models for 10 epochs each using Adam, SGD, and SGD with momentum. Each model should have a single hidden layer with 30 units and ReLU activation. **Todo:** Implement the `create_and_train_model` function to construct and train these models. Then, utilise this function in `compare_optimizers` to evaluate the performance of the different optimizers.

   Run `train_models.py` to run your code.

3) [2 points] **Todo:** Compare Adam vs SGD vs SGD with Momentum.

   Create two plots, one for training loss curves and one for training accuracies curves, for each of the three models above. Complete the implementation of the function `plot_learning_curves` in file `lib/plot.py`.

   Run `plot_trained_models.py` to run your code.

4) [4 points] **Todo:** Implement the learning rate scheduler `PieceWiseConstantLR` and `CosineAnnealingLR`. You can use the provided LambdaLR class for this, which works analogously to the PyTorch LambdaLR. (class `PiecewiseConstantLR` and `CosineAnnealingLR` in file `lib/lr_schedulers.py`).

   PyTorch implements learning rate schedules as wrappers for the optimizer and requires the user to call `scheduler.step()` after each epoch. We've already done this for you in the training loop in function `train` in `lib/training.py`.

   Run `python -m tests.test_piecewise_scheduler` and `python -m tests.test_cosine_scheduler` to test your implementation.

5) [1 point] **Todo:** Plot learning rate schedules.

To verify your implementation, plot the learning rate schedules with number of epochs on the x-axis and learning rate on the y-axis. Implement the function `plot_lr_schedules` in file `lib/plot.py`.

Run `plot_lr_schedules.py` to run your code.

3. Experiments

We will now investigate how ill-conditioning can slow down optimization and two possible ways it can be mitigated using momentum or preconditioning.

For these experiments, we aim to minimize a simple quadratic function, similar to the one presented in the lecture, of the form $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top Q\mathbf{x}$, where Q is a positive definite matrix.

1) [1 point] We will investigate the following cases:

- Condition number = 1, momentum = 0.0, preconditioning = False
- Condition number = 8, momentum = 0.0, preconditioning = False
- Condition number = 8, momentum = 0.35, preconditioning = False
- Condition number = 8, momentum = 0.0, preconditioning = True

Note that for all cases, we use SGD as optimizer with the same learning rate and the same initial point. Finally, we stop after either 500 optimization steps or once we reach a value smaller than 1e-5 for the objective function we are aiming to minimize.

Todo: Run `plot_ill_conditioned.py` for the cases mentioned above.

Example: `python plot_ill_conditioned.py --condition_number 8 --momentum 0.0 --preconditioning` for the last case.

Some of the command line arguments that `python plot_ill_conditioned.py` accepts are:

- `--condition_number`: Condition number of the quadratic function matrix. Default: 1.
- `--momentum`: SGD momentum coefficient. Default: 0.0.
- `--preconditioning`: Apply preconditioning matrix (inverse hessian of the quadratic function matrix) to the gradient. Default: False.

Hint: Run `plot_ill_conditioned.py --help` for more options.

Todo: Explain why the second experiment causes zigzag behaviour and why momentum and preconditioning can mitigate this issue.

Include all the generated plots in your submission.

4. [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.

5. [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 20.11.2024 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.