Due: 30.01.2024 (23:59 CET) | **Deep Learning** | F. Hutter & A. Valada
Points: 16 + 2 bonus | 12th Assignment | WS 2023/2024

To get access to this week's code use the following link: https://classroom.github.com/a/6SLLa_Le

---

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`.

Important: For this specific exercise, a new conda environment must be created. To do so, navigate into the assignment directory and execute the following commands in the terminal:

```
>>> conda create --name neps-env python=3.8 -y
>>> conda activate neps-env
>>> pip install -r requirements.txt
```

This exercise focuses on hyperparameter optimization (HPO) for neural networks. We will:

- Define HPO search spaces.
- Train deep learning models with various hyperparameters.
- Perform automated HPO using Random Search as a black box hyperparameter optimizer.
- Perform automated HPO using HyperBand and PriorBand as a multi-fidelity hyperparameter optimizer.

It's the nature of hyperparameter optimization, that you'll have to train a lot of models, leading to longer execution time in this exercise. In this exercise we'll use NePS (short for Neural Pipeline Search) to perform hyperparameter optimization, as it's a specialized solution for DL. Any feedback is highly valued as NePS is still in active development.

1. Implementing a Deep Learning Pipeline

We will be using the MNIST dataset for our experiments. It consists of 70000 grayscale images in 10 classes, 60000 training images and 10000 test images of size 28 x 28.

PyTorch provides a package called torchvision, which automatically downloads the MNIST dataset, preprocesses it, and iterates through it in minibatches.

1) [2 points] **Todo:** Define a model space (function `get_conv_model` in `lib/conv_model.py`). For this you have to build a deep convolutional model with varying number of convolutional layers for MNIST input using PyTorch. Each layer should have a 2D convolution followed by relu and max-pool operation. The final layer has to be a fully connected layer, followed by the log-softmax.

   You can check whether your implementation is correct by using the test file in `tests.test_convnet.py`.

2) [2 points] **Todo:** Complete the function (`training` in `lib/train_epoch.py`) to train a model for one epoch, see function signature for the parameters that you will need. The function should return the validation error (1 minus validation accuracy) after each epoch. You can use the `evaluate_accuracy` function (defined in `lib/utilities.py`) to get the validation accuracy of the model. Do not forget to set the model to training mode.

   You can check whether your implementation is correct by using the test file in `tests.test_train_epoch.py`.

2. Black-box Optimization with Random Search

1) [1 point] Now, you will do hyperparameter optimization using Random Search. For that you first need to define a hyperparameter space which will contain the hyperparameters and their ranges that we're searching for. More specifically, the hyperparameter space should consist of the learning rate which will be passed to the optimizer, number of filters for the first and second convolutional layers. We encourage you to look at the basic NePS HPO example to see how to define a hyperparameter space without fidelity parameters.

   Todo: Define the hyperparameter space (function `get_pipeline_space` in `lib/rs_pipeline.py`).

   You can check whether your implementation is correct by using the `tests.test_rs_pipeline_space.py`.

2) [1 point] Next, you need to finish the `run_pipeline` function, which will be called multiple times by NePS with different hyperparameter configurations. Here, you have to define a model, optimizer and use the previously completed training function. Keep in mind that the model depends on the number of filters, and the number of filters is determined by the pipeline space.

   Todo: Complete the TODOs of the `run_pipeline` function in `lib/rs_pipeline`.

   You can check whether your implementation is correct by using the `tests.test_rs_pipeline.py`

   Todo: Run `run_rs.py` to start the optimization process.

3) [1 point] Let's further investigate which hyperparameters work well and which do not. For that reason, we create a scatter plot of the learning rate (x-axis) and number of filters (sum over layers, y-axis). Here, we scale the size of the dots on the plot by the error in the last epoch (10 to 100). Furthermore, we plot the error curves (error per epoch) for all configurations in one figure and the incumbents (error of the best hyperparameter configuration) across epochs.

Todo: Run `eval_rs.py` to see the plots.

Questions (include both your answers and the relevant figures in submission.pdf):

- What pattern do you see for the scatter plot? Why might it occur?
- Given the error curves would you expect multi-fidelity optimization or black-box optimization to perform better?

3. Multi-fidelity Optimization with HyperBand and PriorBand

1) [2 points] Here we will use the multi-fidelity hyperparameter optimizer Hyperband, which we saw in the lecture.

   Todo: Define the hyperparameter space (function `get_pipeline_space` in `lib/multi_fidelity_pipeline.py`).

   Note: The hyperparameter space has to consist of number of filters for three layers, learning rate, optimizer (Adam or SGD) and number of epochs which will be a fidelity parameter. We encourage you to look at the NePS multi-fidelity HPO example to see how to define a hyperparameter space with fidelity parameters.

   You can check whether your implementation is correct by using the test file in `tests.test_multi_fidelity_pipeline_space.py`

2) [2 points] **Todo:** You now have to run the hyperparameter search with Hyperband. Complete the TODOs of the function `run_pipeline` in `lib/multi_fidelity_pipeline.py`.

   Keep in mind, that we are also searching for the DL optimizer here. Since, we are going to use a multi-fidelity method, we also provide code for checkpointing the models, i.e. if HyperBand increases the budget for a configuration, instead of training from scratch, we continue where we left off.

   You can check whether your implementation is correct by using the test file in `tests.test_multi_fidelity_pipeline.py`

   Todo: Run the `run_multi_fidelity.py` file with `--searcher hyperband` argument.

3) [1 point] Similar to the Random Search part, we will plot the error curves and the incumbents over time.

   Todo: Run the `eval_multi_fidelity.py` file with `--searcher hyperband` as the argument to see the plots.

   Questions (include your answer and the relevant figure in submission.pdf):

   - Did HyperBand always early-stop the configurations with the worst final performance? Why?

4) [1 point] Now we will run the hyperparameter search with PriorBand.

   When using PriorBand an assumption about the best value for the hyperparamater has to be made. In the NePS library, this is commonly referred to as *default* value which is used as the prior for the run.

   Todo: Run the `run_multi_fidelity.py` file with `--searcher priorband` argument.

5) [1 point] Similar to the Hyperband part, we will plot the error curves and the incumbents over time.

Todo: Run the `eval_multi_fidelity.py` file with `--searcher priorband` as the argument to see the plots.

Questions (include your answer and the relevant figure in submission.pdf):

- Did PriorBand always early-stop the configurations with the worst final performance? Why?

6) [2 points] Lastly, we are going to compare Random Search, Hyperband, and Priorband. Note that before, we ran Random Search on a smaller hyperparameter space, in order for the comparison to be fair, we need to rerun Random Search using the same space as HyperBand.

Todo: Run `run_rs_biggerSpace.py` file to run Random Search on the same search space as HyperBand.

Now that we have the results for all three optimizers on the same search space, we can compare them.

Todo: To compare optimizers, specify them using the `--searcher` argument in the `compare_optimizer.py` file. Atleast two optimizers are needed for the comparison, but upto three optimizers can be compared. (eg. `compare_optimizer.py --searcher1 random_search --searcher2 hyperband --searcher3 priorband`).

Questions (include your answer and relevant plots in submission.pdf):

- Give an interpretation of the plot.
- In this case which of the optimizers was more efficient and why?

4. [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.

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 30.01.2024 (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.