To get access to this week's code use the following link: https://classroom.github.com/a/Rf1G3sTE

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` with the answers and solution paths to all pen and paper questions in the exercise. 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 ($\geq 50\%$) 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.

This exercise focuses on hyperparameter optimization with neural networks. We will:

- Define hyperparameter-configuration search-spaces
- Train deep learning models with various hyperparameters
- Use random search as a basic hyperparameter optimizer
- Use BOHB as an advanced hyperparameter optimizer

It's in the nature of hyperparameter optimization, that you'll have to train a lot of models. Therefore execution time will be longer in this exercise.

1. Coding Tasks

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 structure. The final layer has to be a fully connected layer, followed by the log-softmax.

   You can check whether you implementation is correct or not by using the test file in `tests.test_convnet.py`.

2) [7 points] **Random Search**

   2.1 [2 points] **Todo:** Define the hyperparameter space (function `get_configspace` in `lib/configspace.py`). You learn how to use ConfigSpace to define the hyperparameters, by looking at https://automl.github.io/ConfigSpace/master/API-Doc.html
   Note: In Windows, ConfigSpace installation requires "C++ Build Tools". https://visualstudio.microsoft.com/visual-cpp-build-tools

   Note: The configuration space has to consist of learning-rate, number of convolutional layers (min. 1, max. 2 layers) as well as number of filters for each of these layers. See docstring `get_configspace` in `lib/configspace.py` for more info.

   Hint: The `CS.GreaterThanCondition(conditioned_hyperparameter, lefthand_side, righthand_side)` method might be useful.

   2.2 [2 points] **Todo:** Complete the function (`train_conv_model` in `lib.training.py`) to train a model which is defined by the previous implemented configuration. The function should return the model and the validation error after each epoch. You can use the `evaluate_accuracy` function (defined in `lib.utilities.py`), don't forget to switch between train and eval mode.

   Hint: A single sample from your hyperparameter space is a Configuration. You can use the configuration similar to a dictionary, it supports config.keys(), config.values(), value = config[key], key in config, ... You can iterate a DataLoader to access (data, label) batches. Note: If a condition is not met, the conditional hyperparameter is not included in the configuration.

   2.3 [2 points] **Todo:** Now you have to sample new configurations to train and evaluate them. For that, complete the `run_random_search` function in `run_random_search.py`. Particularly, run `n_random_samples` randomly sampled models for `n_epochs`. Don't forget to store the results `(model, config, val_errors)` for further evaluation (a list of tuples, one for each config). *Hint*: `ConfigSpace` objects have a .sample_configuration() method to sample a random configuration.

   Note: You can check your implementation by using the `tests.test_random_search.py`

   2.4 [1 point] Now we should evaluate the previous runs. Evaluation in hyperparameter optimization can mean two different things: On the one hand, we might only be interested in the model with the best performance. On the other hand, we might want to find the best hyperparameter configuration to then train a model with these hyperparameters (but with e.g. more epochs).

   Todo: Complete the function `best_model_random` in `lib.best_model.py` which returns the model and final validation error of the best model in results.

   Let's further investigate which hyperparameters work well and which do not. For this we run a scatter plot of learning rate (x) and number of filters (sum over layers, y). Here, we scale the size of 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.

   Todo: Run `eval_random_search.py` to see the both plots.

   Questions (answer in pdf):

   - What pattern do you see for the scatter plot? Why might it occur?
   - How could you detect configurations with a low error earlier/faster for the error curve plot? Why could this be problematic?

3) [7 points] **BOHB**

3.1 [3 points] Here we will use the more advanced hyperparameter optimizer BOHB (Bayesian Optimization with Hyperband). Based on TPE and Hyperband, BOHB evaluates configurations on your model with increasing budgets. In the context of Deep Learning, budget can be the number of epochs or the number of training samples. In lower budget evaluations, BOHB can look at more configurations. Full budget evaluations avoid missing configurations which are poor at the beginnning but good at the end (and vice versa). At the start of a run, BOHB samples configurations randomly. After some time, BOHB then uses a bayesian model (based on Parzen Tree Estimators), sampling only promising configs.

This exercise part is based on the HpBandSter examples and the HpBandSter documentation. HpBandSter provides a fast implementation of Randomsearch, Hyperband and BOHB. The optimization can easily be distributed between multiple cores or even multiple computers.

The worker defines the hyperparameter problem which we try to optimize. compute(...) should - for a given configuration and budget - return a loss which the hyperparameter optimizer tries to minimize. In our case, we can use the number of epochs as budget and the validation error as loss. As best practice, we define the configuration space also in the worker.

If you need help, you might get some inspiration from the HpBandSter Pytorch worker example.

Todo: Complete all TODO blocks in `lib.worker.py` It's best practice to do a quick sanity check of our worker with the test in `tests.test_bohb.py`. Please notice that we use a scaled up version of the configuration space (as opposed to RandomSearch) but use the same model space.

3.2 [1 point] We now run the hyperparameter search with BOHB and the worker which we defined above and save the result to disk. Try to understand what happens. HpBandSter allows to start additional workers on the same or remote devices to parallelize the executions, that's why we need to setup some network stuff (nameserver, nic, host, port, ...). If you're interested, you can check this out in the HpBandSter examples, but it is beyond the scope of this exercise.

Note: The code below will try 60 different configurations. Some of them are executed at multiple budgets, which results in about 80 model training. Therefore it might take a while (15-45 minutes on a laptop CPU). If you are interested in how BOHB works, check out BOHB (Falkner et al. 2018). You can see the progress in your command line

Todo: Run the `run_bohb.py` file.

3.3 [1 point] The result object which we dumped to disk contains all the runs with the different configurations. Here we will analyse it further. The HpBandSter analysis example is there, if you need help.

Todo: Print the model of the best run `best_model_bohb` in `lib.best_model`, evaluated on the largest budget, with it's final validation error. *Hint*: Have a look at the HpBandSter result docs.

3.4 [2 points] We can gain deeper insight through plotting results. Thanks to the HpBandSter visualization module plotting is a one-liner.

Note: In order to answer the questions below, run `eval_bohb.py` and interpret the plots one by one.

Note: Results are not deterministic so include the plots in your pdf.

First let's see, if we really can evaluate more configurations when making use of low budget runs. For that, we plotted the finished runs over time.

Answer in pdf: How many runs per minute did approximately finish for the individual budgets?

Evaluating configurations on lower budgets doesn't make sense - even if they are faster - if the performance ranking isn't consistent from low to high budget. This means, that the loss rankings for configurations should correlate. In simplified terms: The best configuration after one epoch should also be the best after nine epochs, the second best should stay the second best and so on.In order to do so, we added a correlation plot of ranking

across budgets.

Answer in pdf: What budgets correlate the least?

We usually assume, that training on a higher budget (number of epochs) and sampling more configurations can lead to better results. Let's check this by plotting the losses over time.

Answer in pdf: Do our assumptions hold true? Why?

Remember that BOHB uses a model after some time to improve the configuation sampling. We can check, if the BO-sampled configurations work better than the random-sampled. To do so, we plot a loss histogram for all budgets only with BO-sampled and only with random-sampled configurations (6 histograms).

Answer in pdf: Is the BO-sampling useful?

2. [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.

3. [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 22.12.2021 (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.