Due: 24.01.2023 (23:59 CET) | **Deep Learning** | F. Hutter & A. Valada
Points: 16 + 2 bonus | 11th Assignment | WS 2022/2023

To get access to this week's code use the following link: `https://classroom.github.com/a/PxWDgu0-`

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 ($\geq 50\%$ 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: When reusing the environment from last exercise, make sure to update NePS to version 0.8.1.

```
>>> pip install --upgrade neural-pipeline-search
```

This exercise focuses on Neural Architecture Search (NAS). It will be based on NAS-Bench-201. NAS-Bench-201 is a neural architecture search benchmark. It consists of a cell search space with 15,625 different possible architectures, and a macro architecture into which the cell is plugged in. Models of each of these macro architectures were trained from scratch for 200 epochs, evaluated, and the results were made available as a tabular benchmark which contains information such as the train/validation accuracies/losses at each epoch, the total time taken to train the model, and the number of parameters in each model.

Additionally, in this exercise we will also use NePS (short for Neural Pipeline Search) to perform Neural Architecture Search by training a model from scratch. Lastly, using NePS we are going to drastically speed up the NAS pipeline by querying NAS-Bench-201 instead of training an architecture from scratch.

More specifically In this exercise, we will:

- Implement a NAS pipeline using NePS, on the cell search space using a simple fixed macro-architecture and train each architecture from scratch, using Bayesian Optimization as the search strategy.

- Extend this NAS approach to the NAS-Bench-201 search space.
- Perform joint Hyperparameter Optimization and Neural Architecture Search (JAHS) using NePS.
- Implement a benchmark API to query NAS-Bench-201.

Like hyperparameter optimization, NAS is also computationally quite expensive. To keep the assignment accessible to everyone, we will use only 5% of the dataset for training the models. Further, we will train them for only 5 epochs each during the search phase of the optimizer.

1. Coding Tasks

1) [2 points] **Warmup:**
Similar to the HPO exercise, we will begin by implementing the training function that we will use to train the models.

In this exercise we will also make use of the `torchvision.datasets` which provide a collection of freely available image datasets. You can find more information on `torchvision.datasets` in the documentation page.

We will also use PyTorch `Dataloader` class. This class returns an object that we can use to iterate over the dataset. A more detailed description can be found in the documentation page.

Todo: Complete the TODO blocks in `lib/training.py`.

Run `tests/test_train.py` to test your implementation.

2) [3 points] **Simple NAS pipeline with NePS:**
Let us begin by familiarizing ourselves with how to create a simple NAS pipeline with a cell search space and search this space using Bayesian Optimization with NePS.

The cell we will use consists of 3 nodes with 3 edges as shown in Figure 1 (the connection of edges is handled internally by NePS).

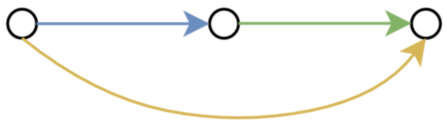

Figure 1: A cell of the simple NAS pipeline. Each edge can hold a different operation.

Each edge can hold any one of the following operations:

- `identity operation`
- `3x3 convolution`
- `3x3 average pooling`

The fixed macro-architecture that we will use for our simple pipeline is as follows:

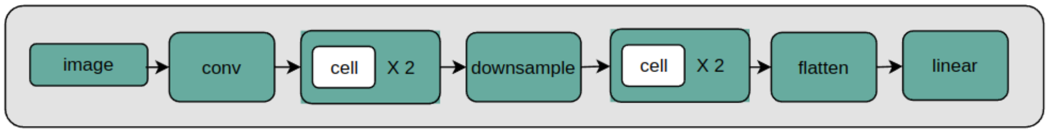

Figure 2: Fixed macro-architecture of simple NAS pipeline.

We will train each pipeline from scratch, on a small subset of CIFAR-10, using a batch size of 64 and a learning rate of 10^{-3} for a total of 5 epochs.

Todo: Implement the simple NAS pipeline (`run_nas_simple.py`) and execute the script to run the pipeline.

Run `tests/test_run_nas_simple.py` to test your implementation.

After implementing this, you can run `eval_simple.py` to see the error curves and incumbents plot of the different models.

3) [4 points] **Extended NAS pipeline with NePS:**
Now we will move to a more complicated cell search space and macro-architecture. More specifically, we will consider the NAS-Bench-201 cell search space. Here, each cell consists of 4 nodes with 6 edges as shown in the figure below. Edges hold operations, and the nodes are used for summing up the tensors from the incoming edges.

Each edge can hold any one of the following operations:

- `zeroize operation`
- `identity operation`
- `1x1 convolution`
- `3x3 convolution`
- `3x3 average pooling`

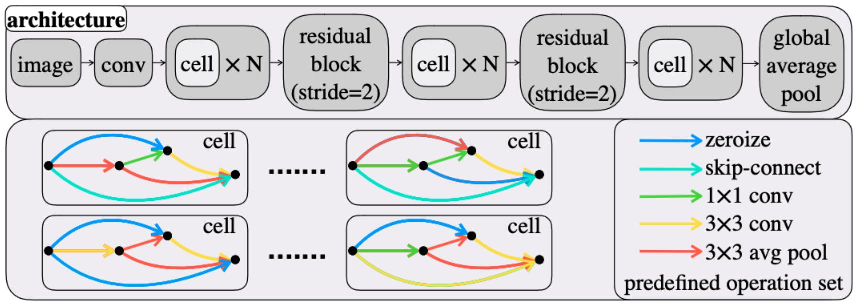

Figure 3. **Top**: the macro skeleton of each architecture candidate. **Bottom-left**: examples of cells. Each cell is a directed acyclic graph, where each edge is associated with an operation selected from a predefined operation set as shown in the **Bottom-right**.

Similar to the simple NAS pipeline considered previously, we will train each pipeline from scratch, on a small subset of CIFAR-10, using a batch size of 64 and a learning rate of 10^{-3} for a total of 5 epochs.

Todo: Implement the NAS pipeline that follows the NAS-Bench-201 architecture (`run_nas.py`) using $N = 5$ and execute the script to run the pipeline.

Run `tests/test_run_nas.py` to test your implementation.

After implementing this, you can run `eval_nb201.py` to see the error curves and incumbents plot of the different models.

4) [4 points] **Joint Architecture and Hyperparameter search with NePS:**
Using NePS we can jointly optimize both the architecture and the hyperparameters (JAHS). In that direction, in addition to the architectural parameters, we will now also consider the learning rate, batch size, and choice of optimizer for our pipeline.

Todo: Implement the joint HPO and NAS pipeline using NePS (`run_joint_nas_hpo.py`)

Run `tests/test_run_joint_nas_hpo.py` to test your implementation.

After implementing and running the joint HPO-NAS pipeline, you can run `compare_methods.py`, to see a comparison plot between joint HPO-NAS and NAS.

5) [3 points] **NAS-Bench-201 API:**
We will now implement a small API to query the results from the NAS-Bench-201 experiment, which trained all the 15,625 models in the search space for 200 epochs and saved the results in a `.pth` file. The original file is quite heavy (> 2 GB), so we've made a lighter version of it for this exercise (111 MB). Please download the file named *nb201_cifar10_full_training.pickle* from here and put it in the `benchmark` folder.

Let us first familiarize ourselves with the structure of data in *nb201_cifar10_full_training.pickle*. It is a dictionary with the following format:

```
{
    "<string_representation_of_architecture>": {
       "cifar10-valid": {
             "train_acc1es": [...], # List of top-1 train. accuracies for each of the training epochs
             "eval_acc1es": [...], # List of top-1 val. accuracies for each of the training epochs
             "train_losses": [...], # List of training losses for each of the training epochs
             "eval_losses": [...], # List of validation losses for each of the training epochs
             "cost_info": {
                 "train_time": value, # Time taken to train this model, in hours
                 "params": value, # Number of parameters in this model, in millions
             }
        }
    },
    .
    .
}
```

Todo: Implement the NAS-Bench-201 API (`lib/benchmark_api.py`):

Run `tests/test_benchmark_api.py` to test your implementation.

Finally, complete the code in `run_api.py`, and run the script. The following code, will evaluate 50 models instead of 10 as in the previous tasks.

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 24.01.2023 (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.