Due: 13.01.2021 (23:59 CET) | **Deep Learning** | F. Hutter & A. Valada
Points: 16 + 2 bonus | 10th Assignment | WS 2021/2022

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

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 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 queryable 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.

In this exercise, we will:

- Implement a benchmark API to query NAS-Bench-201.
- Implement the NAS-Bench-201 cell search space and its macro graph in PyTorch.
- Implement an optimizer to search this space.
- Run this optimizer with and without the use of the benchmark API.

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 10 epochs each during the search phase of the optimizer.

You will see a new library, `networkx`, being used in this exercise. `networkx` is a package which can be used for creation and manipulation of graphs in Python. Make sure you have it installed before starting the exercise.

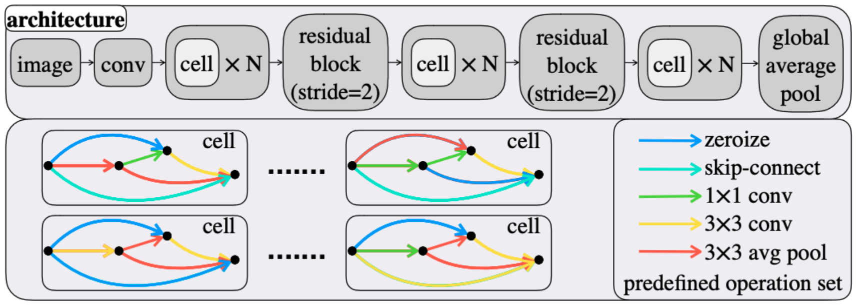

Figure 1. **Top**: the macro skeleton of each architecture candidate. **Bottom-left**: examples of cells. Each cell is a directed acyclic graph, where each edge is associated with an operation selected from a predefined operation set as shown in the **Bottom-right**.

1. Coding Tasks

Let us begin by familiarizing ourselves with the NAS-Bench-201 search space. It consists of a cell with 4 nodes, and 6 edges, as shown in the figure above. 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`

We shall implement 'same' convolutions, so that the dimensions of the input and output tensors of a given edge are the same. For average pooling too, we shall use the appropriate stride and padding so that the output dimensions are the same as the input dimensions.

The `NASBench201MacroGraph` (architecture shown at the top in the figure above) is obtained by using `Stem` for 'conv', `NASBench201CellSearchSpace` for the cells, and `ResNetBasicblock` for residual blocks. All of these classes are provided for you in `lib/primitives.py`.

At each residual block, the height and width of the input tensor is reduced, while the number of channels is doubled.

Since there are five choices of operations for each of the six edges, the final search space will have $5^6 = 15,625$ architectures. In this exercise, we shall implement a simple random search optimizer, which randomly samples cell configurations and evaluates the resulting `NASBench201MacroGraph`.

1) [2 points] **Todo:** Complete class `CategoricalOp` (`lib/categorical_op.py`)

   Every edge in the cell graph will hold an instance of `CategoricalOp`, which in turn can hold one of the five different operations based on its instantiation.

   Run `tests/test_categorical_op.py` to see if your implementation is correct.

2) [8 points] **Todo:** Define the cell search space (`lib/cell.py`):

   `NASBench201CellSearchSpace` inherits from `torch.nn.Module` as well as `networkx.DiGraph`, so we can combine the functionalities offered by both these classes.

   Complete the methods in the following order:

- `get_configuration_space()`
- `_build_graph()`
- `forward(...)`
- `get_string_representation()`

You can refer to the docstrings of these methods for more details about them.

Run `tests/test_cell.py` to see if your implementation is correct.

3) [2 points] **Todo:** Create the macro graph (`lib/graph.py`):

Complete the code to create the macro graph as shown in the Figure 1 (at the top).

Run `tests/test_macro.py` to see if your implementation is correct.

4) [2 points] **Todo:** Implement the optimizer (`lib/optimizer.py`):

Complete the `search(...)` method in `RandomSearchOptimizer`. In each iteration, the optimizer must sample a random configuration of the cell, instantiate a `NASBench201MacroGraph` using it, and then train and evaluate the model. Also, keep track of the configurations and the accuracies of the models at the end of each iteration.

Run `run_optimizer.py` to search the `NASBench201CellSearchSpace` using the `RandomSearchOptimizer`. To keep the training time minimal, we will only sample 5 models, and train them each for 10 epochs using only 5% of the training data. Naturally, the performance of these models will suffer for being trained so little. We will see how we can use the NAS-201-Benchmark to overcome this issue in the next section.

You can play with using higher number of epochs or bigger portions of the training dataset by tweaking the code in the `train(...)` function in `lib/training.py`, but please make sure to revert it before making your submission.

5) [2 points] **Todo:** Implement the NAS-Bench-201 API (`lib/benchmark_api.py`):

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 training accuracies for each of the training epochs
             "eval_acc1es": [...], # List of top-1 validation 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
             }
        }
    },
    .
    .
    .
}
```

Complete the code in `lib/benchmark_api.py`. Once you're done, you can run `tests/test_benchmark_api.py` to test your implementation. You can also check out the results you can get for a single model from the benchmark by running `plot_curves.py`.

Finally, complete the `evaluate(...)` method in `RandomSearchOptimizer`, and run `run_optimizer.py` with command line parameters `--use-api --n-iter 100` . This should force the `RandomSearchOptimizer` to use the benchmark api to query the performance of a given architecture, instead of training it from scratch. Additionally, it will also make it sample 100 models instead of 5.

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 13.01.2021 (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.