General constraints for code submissions Please adhere to these rules to make our and your life easier! We will deduct points if your solution does not fulfill the following:

- If not stated otherwise, we will use exclusively Python 3.5.
- If not stated otherwise, we expect a Python script, which we will invoke exactly as stated on the exercise sheet.
- Your solution exactly returns the required output (neither less nor more) – you can implement a `--verbose` option to increase the verbosity level for developing.
- Add comments and docstrings, so we can understand your solution.
- (If applicable) The `README` describes how to install requirements or provides addition information.
- (If applicable) Add required additional packages to `requirements.txt`. Explain in your `README` what this package does, why you use that package and provide a link to it's documentation or GitHub page.
- (If applicable) All prepared unittests have to pass.
- (If applicable) You can (and sometimes have to) reuse code from previous exercises.

In this exercise you will learn how to run your NAS optimization algorithms on NAS-Bench-101. To avoid that you waste too much time training the sampled architectures, we will not optimize the actual benchmark but instead use a tabular benchmark. Refer to "NAS-Bench-101: Towards Reproducible Neural Architecture Search" for more details about the search space and encoding.

In NAS-Bench-101 every CNN architecture in the cell search space was trained and evaluated 4 independent times. This was done for 4 different budgets (training epochs), however here we will only use the results for the highest budget (108 epochs). The results are compiled into a database, such that we can simply look up the performance of a hyperparameter configuration instead of training it from scratch.

1. **Getting started with NAS-Bench-101** [1 point]

   `NASBench_example.py` shows how we can load the benchmark together with the configuration space. Before running this script make sure you have installed all the dependencies (including tensorflow, pytorch, nasbench, etc.) as shown in `setup.sh`. To execute the setup run `bash setup.sh`. Each hyperparameter configuration is encoded as a `Configuration` object and we can easily convert it to a `numpy` array or a `dictionary`. Note that, if we convert it to a numpy array, all values are normalized to be in $[0, 1]$.

   If we evaluate a hyperparameter configuration, we get the validation error and the time it had taken to train this configuration. When we generated this benchmark, we evaluated each hyperparameter configuration 4 times, and, for every table lookup, we pick one of these 4 trials uniformly at random. This simulates the typical noise that comes with hyperparameter optimization problems.

   Here you will just have to run `python NASBench_example.py > logs/log.txt` and push the resulting `log.txt` to your Bitbucket repository.

2. **Blackbox optimization on NAS-Bench-101** [7 points]

   In this exercise you will have to run the command `python main.py` which will execute Random Search (RS), Regularized Evolution (RE) and Non-regularized Evolution (non-RE) on NAS-Bench-101. But before running this command you will have to fill the missing code in `optimizers.py`.

   (a) In the `NASOptimizer` base class you will have to implement two methods which return a sampled configuration (architecture in this case) from the configuration space, and that query the validation + runtime of a specific configuration and update the incumbent trajectory. [3pt.]

   (b) Afterwards, you will have to implement (non-)RE based on the pseudocode presented in Slide 24, Lecture 7. RE keeps a population of architectures and each time it does an update, it discards the oldest architecture configuration present in the current population. On the other hand, non-RE discards the worst architecture. Everything else is the same as RE. [4pt.]

   Now you are ready to run `python main.py`. It runs by default RS, RE and non-RE by default 20 independent times for 100 function evaluations (`n_iters`). You may change their values if you want.

After the optimization is finished, a plot `logs/plot.png` will be generated. You will have to push this plot together with `optimizers.py` in your Bitbucket repository.

NOTE: `main.py` should run on a CPU machine in less than a minute (with the default arguments).

3. **Feedback** [Bonus: 0.5 points]

For each question in this assignment, state:

- How long you worked on it.
- What you learned.
- Anything you would improve in this question if you were teaching the course.

This assignment is due on 05.07.19 (10:00). Submit your solution for the tasks by uploading a PDF to your groups BitBucket repository. The PDF has to include the name of the submitter(s).