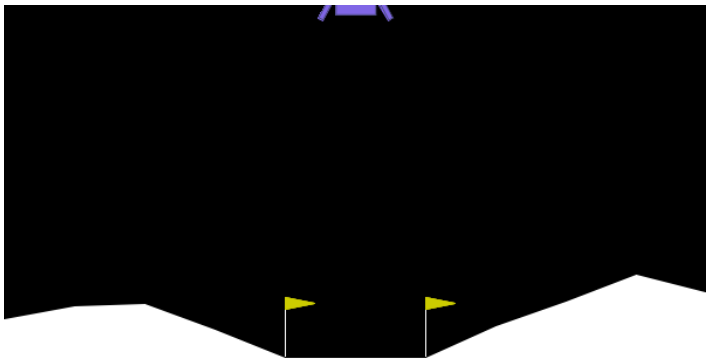
InstaDeep RL/GNN internship 2023 - Berlin

Congratulations on succeeding at the first technical interview!
Welcome to the second phase of the recruitment process, which consists of a RL take-home test. You will find two challenges in this document, a RL-challenge and a GNN-challenge. You are expected to choose:

- The first challenge **“Lunar Lander Environment "** if the chosen topic is multi-objective optimization for train scheduling and re-dispatching.
- The second challenge **“MiniGrid loadflow solver”** if the chosen topic is GNNs to optimize power grids.

You are not expected to solve both!

Challenge: Lunar Lander environment



Snapshot from the Lunar Lander environment, source Gymnasium official docs

Environment Definition:

All the details about the environment can be found in the official Gymnasium documentation.

InstaDeep™

Your tasks in this project:

The goal of this project is to implement a deep RL solution to solve the Lunar lander Environment.

First task: Interact with the environment:

Initially, we use the default environment version with discrete action space.

- Get yourself familiar with the env APIs.
- Play an episode with random actions and visualize the Lander failing to land safely.

Second task:

Implement a Deep RL solution to solve the Lunar Lander environment.

- You can implement your own agent or use a RL framework.
- Using the **Ray/RLlib** framework is preferable but is not required.

The solution should include:

- The code used to train the agent.
- Learning curve plots [episode reward for instance or any other relevant metrics to follow the agent progress]
- A **checkpoint** of the trained agent.
- Two entry points ***train.py*** and ***eval.py***. The first one is to be launched to train the agent and the second one is to run a rollout using the saved checkpoint.

Third Task: Prepare deliverables

- A **presentation** in **pdf** format explaining the implementation details, justification of choices (agent, neural network architecture, Hyperparameters, etc) and the ideas of future work.
- A **Readme.md** detailing how to use the repository to reproduce the presented results.

InstaDeep™

Fourth task (Bonus):

1. Compare your results with a heuristic based solution. You can implement your own version or use the one from Gym.
2. Explore tweaking the reward to make the lander land safely with **less fuel consumption**. Investigate how this change affects the solution using custom metrics or the renderer.
3. Explore your agent skills with the continuous version of the environment. Investigate the necessary changes and try to solve the environment.

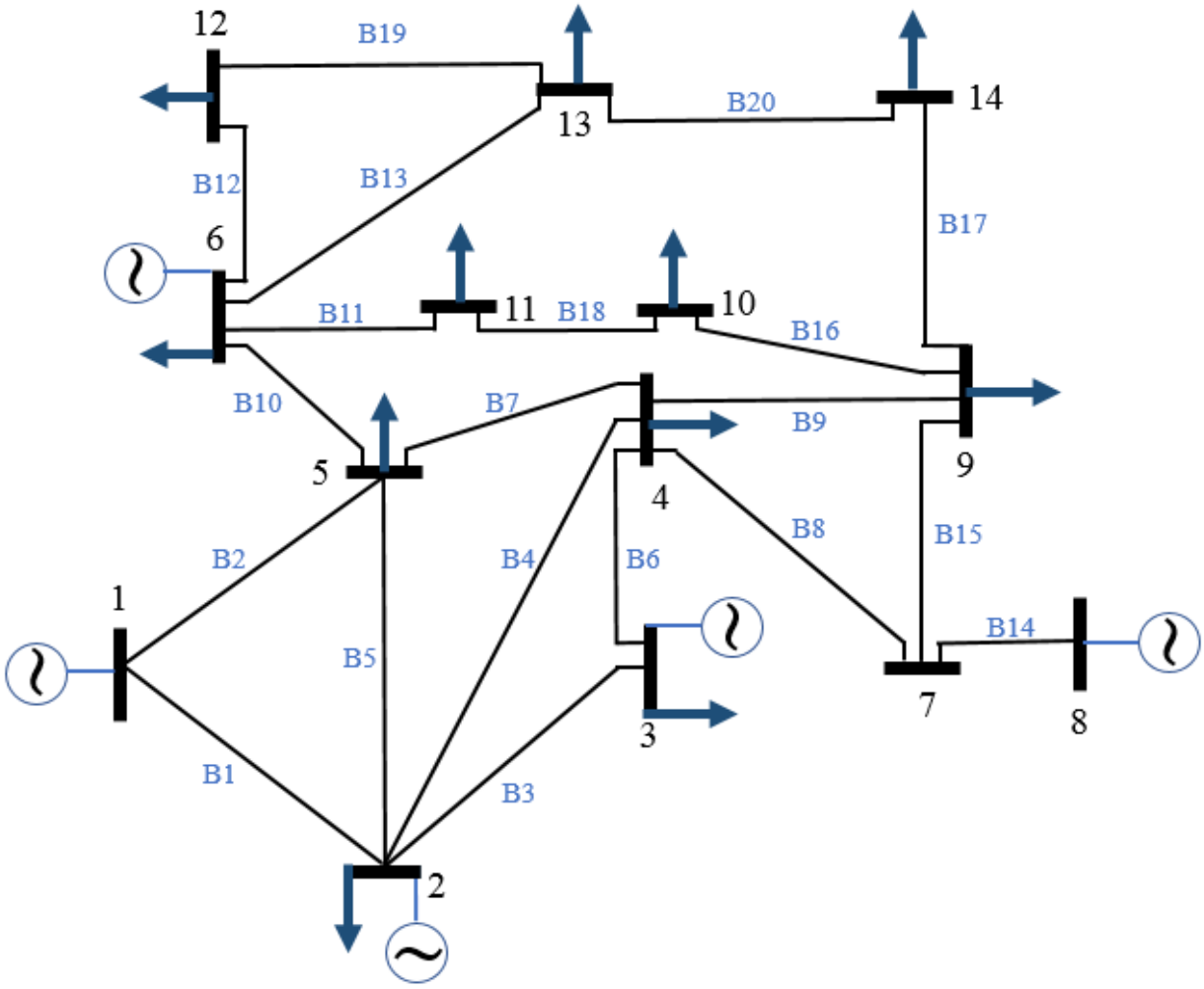
General Guidelines:

- A convergent agent does not mean a successful project. We emphasize more on the **code quality , readability, and the solution design.**
- Document your progress iteratively and explain what made the agent fail or succeed at the task.
- Don't panic if your agent does not learn. It does not mean that you failed the project. Focus on being pedagogical in your deliverables and clearly explain your choices.
- Consider making your input configurable by leveraging yaml files.
- Use the framework **(tf or pytorch)** that you are familiar with the most. Rllib supports both frameworks.

- There are open source examples on the usage of Rllib. They can be useful to design your solution.

The solution should be shared using a **private Github repository** to this account (github id: KhalilGorsan)

Challenge: Minigrid loadflow solver

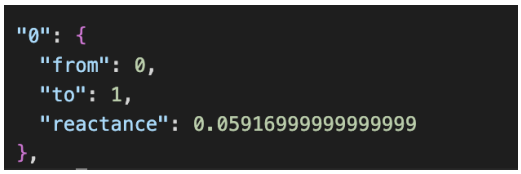


Depiction of the IEEE case14 powergrid, source

Description

Download the dataset from this URL - this file contains a simplified description of an electricity grid. This grid has 14 busbars and 20 branches that connect these busbars. In this case, a busbar is the electrotechnical equivalent of a substation, where multiple powerlines connect, and a branch is the equivalent of a powerline that is built between substations to transport energy. We simplified the grid even further from its original state by using the DC approximation, joining all nodal injections and pre-computing the injections at the slack bus.

You can get information about the branches in the "**adjacency.json**" file, where you will see the two busbar indices in the "from" and "to" field that this branch connects and its reactance.



Branch 0 for example connects busbar 0 (from-end) and busbar 1 (to-end), though in our case the direction doesn't matter - electricity can flow both ways and is equally hindered by the reactance, in this case 0.059 p.u.. Note that in the picture above, busbar and branch indices start at one and in the dataset they start at zero.

Furthermore, there is an **injections.npy** file which holds a (n_timesteps, n_bus) numpy array. Each timestep holds the amount of power in MW that is injected onto every bus. In our simplified network, this single number is everything you need to know about a busbar to compute the loadflows. The injection can be positive or negative. Positive means that there is net energy added into the system at this place, usually by a generator. Negative means there is energy drawn from the system at this place, usually by a load. Zero means that this bus is just a transit bus, and power flows through it to connected branches.

The **loads.npy** file is the target that we want to predict. This is a (n_timesteps, n_branch) numpy array that holds the current on each branch. As each branch has a maximum rated current that it can safely transport, we need to know the current that is expected on every branch for a given injection vector to check for possible overloads.

First Task

Train a GNN to approximate the branch loads. You are free to use any open-source toolsets you desire to achieve this task. It's not important that the GNN converges or performs well, rather we expect you to be able to explain your engineering decisions: Which neighborhood aggregation to use, which metrics you measure, how you chose to architect the neural network, etc.

The solution should include:

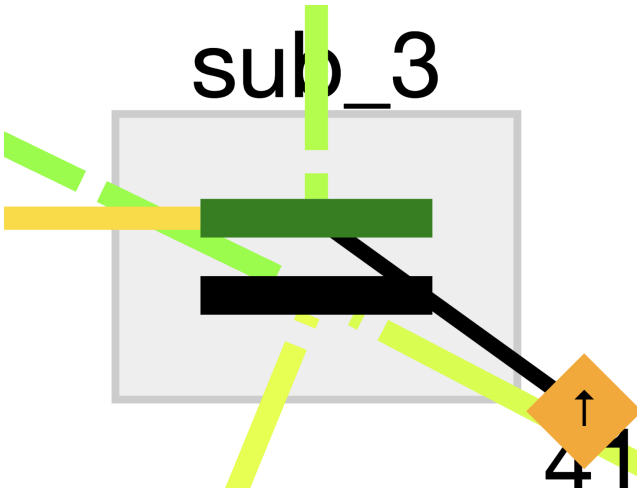
- The code used to train the model.
- Learning curve plots [for whichever metrics you deem relevant]
- A **checkpoin**t of the trained model.
- Two entry points ***train.py*** and ***eval.py***. The first one is to be launched to train the model and the second one is to obtain a **loads.npy** file from a given **injections.npy** and **adjacency.json** file.

InstaDeep™

Second Task:

Underestimating the loads is far more dangerous than overestimating the loads. Measure the bias of your model and think of ways how you could potentially reduce underestimation at the cost of more overestimation. You don't have to implement this, just document your thoughts.

Third Task:



Busbar 3 has been split in two by the grid operator who decided that a split configuration is beneficial to operating the grid for the next 50 timesteps. Branch 3, 15 and 16 are on one side of the split, and branch 5, 6 and the load are on the other side. Hence, what used to be busbar 3 now needs two busbars to be electrically represented. A new busbar 14 has been added to the gridmodel to accommodate this new situation. The data to this can be found in validate.zip. Validate your previously trained model on this new situation.

General Guidelines:

- A convergent model does not mean a successful project. We emphasize more on the **code quality , readability, and the solution design.**
- Document your progress iteratively and explain what made the model fail or succeed at the task.
- Don't panic if your model does not learn. It does not mean that you failed the project. Focus on being pedagogical in your deliverables and clearly explain your choices.
- Consider making your input configurable by leveraging yaml files.
- Use the framework that you are familiar with the most.

The solution should be shared using a **private Github repository** to this account (github id: blacksph3re).