PROJECT REPORT

Group : 1bis/3/4
Team : Grid
Members : BEREUX Nicolas, COLIN Jonathan, PONTIGGIA Florian, RENOUARD Gwenn, SOULARD Thibaut, RATOVO Kevin
Challenge URL : https://codalab.lri.fr/competitions/398
Last submission ID : 7543
Video presentation URL : https://www.youtube.com/watch?v=Pz46eIaSAqw
Github repository : https://github.com/GridL2RPN/Grid
Presentation slides :
https://github.com/GridL2RPN/Grid/blob/master/MiniProjet%20_%20Slides%20pr%C3%A9sentation.pdf

I – Introduction

You've all lived that moment : You're watching TV, a video, perhaps a movie on your phone or computer. Perhaps, something is cooking in the oven at the same. And then suddenly, the power goes out. These outages can happen for various reasons, for example faulty material on some power lines. But sometimes it's due to wrong management, with lines overflowing or not receiving enough power. Right now, power management is handled by humans, so they're bound to sometimes make mistakes. That's where our project comes in.

The L2RPN challenge, short for Learn to run a power network, is reinforced learning project. The goal of this project is for our agent to be able to run a power network on its own, that is provide all the energy needed for every load considering what is being produced by the different substations. Reinforcement learning is based on 4 basic entities : agents, states, actions and rewards. Here, the state of the electric grid represents our different states : The number of loads, the power received by each of them, the power running through our lines, the configurations of our lines regarding substations… etc. Actions are a bit simpler; our agent can switch on or off the different lines and rearrange them with node splits. Rewards are a numerical value, representing how well our agent is doing (see bonus pages). Their computing takes many parameters into account. Obvious ones such as if the power requirement is met for loads, but also more subtle ones like how different the actual grid is from the initial one.

We are provided with a unique module called pypownet, which contains numerous functions to not only manage a network model but also visualize it (Fig 1). We're also provided with 3 different data sets by difficulty : easy, medium and hard each containing the daily consumption (Fig 2) and production of power plants (Fig 3), maintenances planned, and eventual hazards on the power grid. And on top of that, all of this is split into 11 chronics, represented by each different curve on the figure 2, each of them having different power needs. This means that we'll need either to have an agent for each of those chronics or learn on all of them. Seeing the similarities in the shape of all the curves, we can probably have good results on each of them if we train of each of them.

We've each chosen this project for our own reasons. The most common one was that reinforced learning seemed more interesting. We find projects on neural networks very similar to each other, so we wanted to try something new and exciting. Another thing to note is that, while we find RL more interesting, we also find it harder and wanted to challenge ourselves. We might have bitten more than we could chew, as this project was very hard to handle from start to the end.

Our team is comprised of 6 people, divided into 3 teams of 2 persons :

Pre-processing : Pontiggia Florian and Renouard Gwenn.
Learning Agents and Algorithms : Béreux Nicolas and Colin Jonathan.
Visualization : Ratovo Kevin and Soulard Thibaut.

We found the workload expected from the project very unevenly distributed depending on which team you were put on. So, we've settled on letting the visualization team doing everything unrelated to code plus producing graphs, while the other teams focus on their own part. Of course, we still had everyone give their ideas and help each other when a bug showed up or if we needed input from somebody else.

We will now move onto how we approached the challenge.

II – Algorithms, Pseudo-code and obstacles

Throughout this project, we've come up with a lot of ideas of agents. We expected the project to be more in line with what we've done the previous semester in Info232, so we struggled to produce something relevant in the first weeks. Our very first agent didn't even qualify as a learning agent. It was a determinist algorithm (*Algorithm 1*). The idea of this agent was to randomly turn on an inactive line on every iteration. And once a line's power usage rises over 80% of its capacity, we turn it off. We expected it to have a bad score, but it would allow us to learn how to manipulate many pypownet methods, making the time spent worth it regardless of results. Our agent's code was tested through prints and score checking.

While the results of our first agent weren't that bad, we knew it wouldn't cut it for a final submission. So, this time, we made our own Greedy agent, (we'll call it GreedySub) which is based off the agent given in the baseline_agents.py of pypownet. It was specifically designed to score better than DoNothing, choosing actions only if its reward is equal or greater than the DoNothing agent. Outside of this subtlety, GreedySub is a very classic Greedy agent (*Algorithm 2*). For each iteration, it looks at its available actions and computes the reward for each of them and then chooses the best one. However, considering the size of the array representing our actions, searching through every action would be very slow. So, we chose to only consider actions where 1 or 0 switches are turned on/off. We only needed to make sure that actions taken were a single switch press, and that the returned action was indeed the best, so this agent was tested through prints and asserts.

Even with this compromise made, this submission on Codalab took over 20 minutes to execute itself on a single chronic. This isn't surprising, as while this agent is closer than the first one, it is still not a learning agent. It searches for the best action but doesn't associate it with the state it's taken from : it doesn't remember what he learns. And even if we did store the best action, it wouldn't matter as the states in our challenge are represented by arrays that contains floats representing the usage of each line. While a 10^{-2} difference between two states isn't likely to change the best action, it would prevent the states from being identified as the same. So, while this agent's results were quite satisfying, we could still optimize it.

We've established that we have 2 problems to solve : the speed of our algorithm and regrouping similar states to allow our agent to learn. That's where the pre-processing comes in.

There are multiple ways to solve both problems. For the speed problem, we've decided to build an imitation agent. Instead of having our agent search for the best actions on every run, we'll first have a Super agent do all the work and our agent will simply follow the policy built by the Super agent. While running the Super agent still is time expensive, it only needs to be ran once.

For the state problem, we've decided to implement a neural network. It will classify states into different categories, where each state's category corresponds to the best action available from this state. On each iteration, the state encountered will be classified by the network, and the agent will take the action given in return. This solves the problem of states being close enough but not recognizable by the agent. The data used to train and test the network is taken from a run from the Super agent. As there are already many pre-built methods for neural networks, we've decided to not write our own code but to run our imitation agent with different networks (*Fig 4*). It was narrowed down to the MLPClassifier from the sklearn.neural_network package and SVC from the sklearn.svm package. However the MLPClassifier is way faster while having very similar results so we kept it. This method takes a lot of optional parameters in, only a few that we really understand. We mainly ran the method changing parameters one by one, and ended up picking the default parameters except for the learning rate now set on "adaptive" and the activation function set on logistic. 90% of our data is used as the training set, and 10% as the test set as seen in the usage of default parameters.

The Super agent we've chosen to implement is a GreedySearch agent, which is also a slightly modified version of the one provided by pypownet, except that it is now ϵ-greedy (*Algorithm 3*). This marks the appearance of ϵ, our first hyper-parameter. We randomly generate a number between 0 and 1. If this number is strictly smaller than epsilon, our Greedy agent will randomly split a node. If it's between ϵ and 2ϵ, our Greedy agent will randomly switch on or off a line. Else it will do the same thing as our previous Greedy agent : look for the best action where only 1 switch or none are pressed and take that action. Having the RandomNodeSplitting agent as a disruptor is a key component of our imitation agent. While the Greedy agent will avoid taking these actions, they are still available to him, so not only do we need our agent to be able to recover from these actions, but we also need to test actions in a more exhaustive way. We initially only disrupted our agent with the RandomNodeSplitting agent, but we figured this prevented our agent to keep on visiting a set configuration of a grid, so we added the RandomLineSwitch agent in the mix. We've settled on having $\epsilon = 0.1$. It's a very classic value for such a parameter and is the one returning the best results for our greedy agent (Fig 5). This is because the disrupt rate is double of epsilon, meaning that anything above 0.15 probably won't give great results, and anything lower than 0.1 would contradict the idea of having a reasonable amount of exploration. Since the code was heavily based off the given agents, we didn't need to test it.

III - Results

We've kind of spoiled most of the results in our analysis, but we will still go over our agents different performances.

We've put up several graphs comparing every agent we've come up with (*Fig 6*). Since every agent is present on this graph, we'll go through them in a chronological way. Our first agent's results were the most surprising, it managed to outscore DoNothing even though it's not a learning agent. It does make sense though, as most of the actions taken by Determinist are DoNothing actions. Every watt of power it gives out can only increase the reward, and it can't do a gameover since it automatically turns off lines.
GreedySub appears to be our best agent. It's policy is actually quite similar to Determinist, he takes a lot of DoNothing actions but optimizes the other action he takes, leading to an even better score.

At the same time as making our Imitation Agent, we came up with a QLearning agent, as it wasn't much effort since this is what we've studied in Info215 (Artificial Life). Its score is very bad, but that's because the QLearning is the type of agent that suffers the most from this challenge's problems. There's no way to visit all states, so there's no way to build a reliable policy. This means that the agent isn't using the things he's learning, hence why the very low score.
And now onto the Imitation agent. On this graph, it appears to be a terrible agent. But, if you've looked at our previous figures such as Figure 6, you can see that it can peak at around a cumulative reward of -100. And each time we re-run the agent, it's score is different and varies heavily (*Fig 7*) even if we're not changing the policy built by our super agent. We initially thought it was caused by the looping of chronics, but even after making those static, the results still change arbitrarily. The only possibility left is that this divergence is caused by the classifier we're using. Since our classifier is probabilistic and we have 53 classes, we can suppose some states end up being classified differently depending on the run.

This isn't the only problem that our imitation agent is encountering. We've built a histogram counting the different actions taken by our GreedySearch (*Fig 8*). GreedySearch does integrate random functions in its implementation, but it's still oddly inconsistent (*Fig 9*). We expected it to take actions in a similar fashion, but it doesn't. But the most odd thing about it is that out of all those actions, none of them are the DoNothing action even though it's one of the most useful ones. This ultimately shows that the problem we're looking for is not in our reasoning but in our code.

IV - Conclusion

This challenge, even if quite hard, turned out to be interesting. While our agents aren't very successful, we discovered a few learning methods that we didn't know about. We do however realize that even if our agent did work, it wouldn't be the best either. There's a big con in using an agent based on greedy-searching : it doesn't look further than one action. That means that actions aren't necessarily the best in the long term. Of course, considering the size of the observation of the grids, we can only search that deep while having a reasonable computing time. One thing we could have implemented is a system used by the other L2RPN group : a credit system. We introduce a currency system used by our agent, and he must pay an amount of this currency to use the greedy agent. Else he takes a random action. If the greedy-search agent is performant enough, our agent should end up in a spot where only good actions are available which would mitigate the punishment for taking a random action.

On the more social side, this was one of our first projects in 'full' autonomy. Knowing that, we should have expected it to be very complex and time consuming. The inclusion of pypownet slowed down the process of understanding the project, and many submissions were done in panic. We don't think we lacked organization within the group but rather with how to handle the project itself. We didn't plan out in advance which parts of the project to take care of and simply followed the instructions given on the website. We also agreed at the start of the project to respect the law of encapsulation. This turned out to be a bad idea, since we didn't all work directly on an agent. It's unsatisfying that the "solution" to the challenge was given to us by teachers, even more knowing that we didn't succeed in implementing it. But we know that we wouldn't have found the solution by ourselves. So, our final lesson and advice would be to not hesitate to ask for help, and most importantly to follow the leads you're given.

Figures

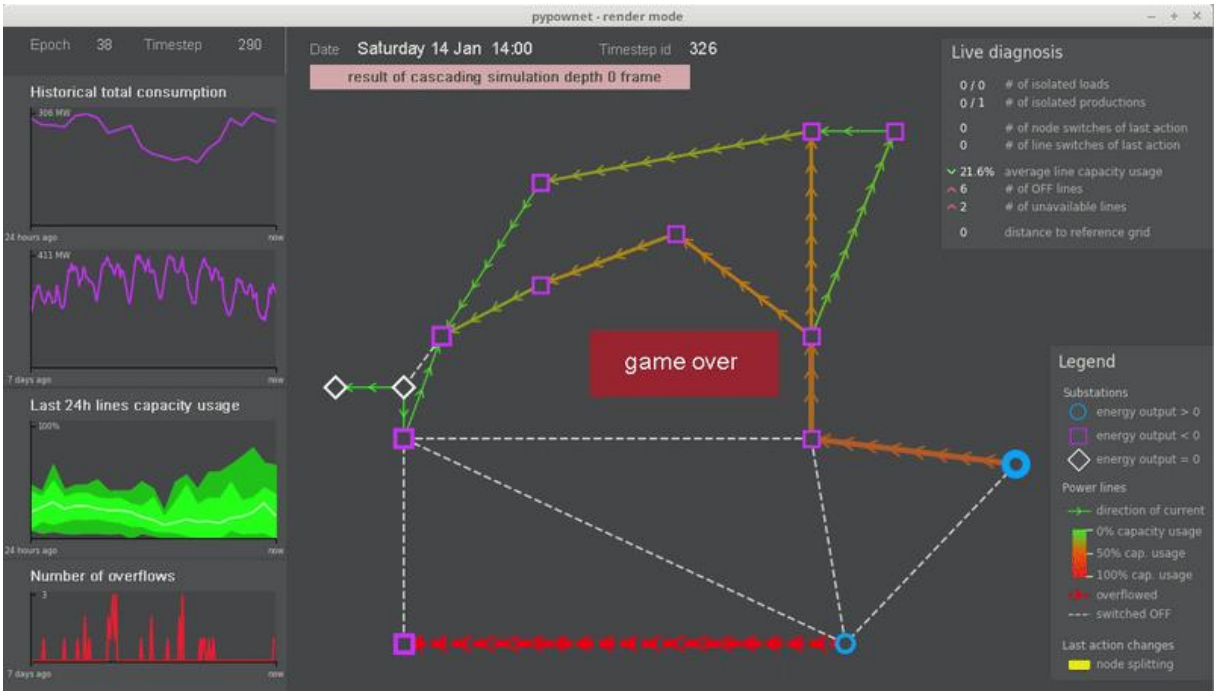

Figure 1 : Example of a grid visualized by pypownet's renderer

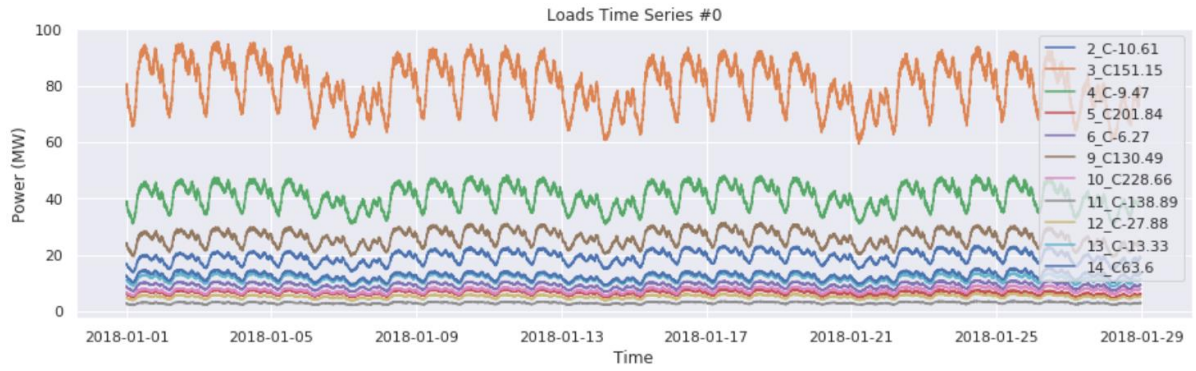

Figure 2 : Daily consumption of loads of different scenarios

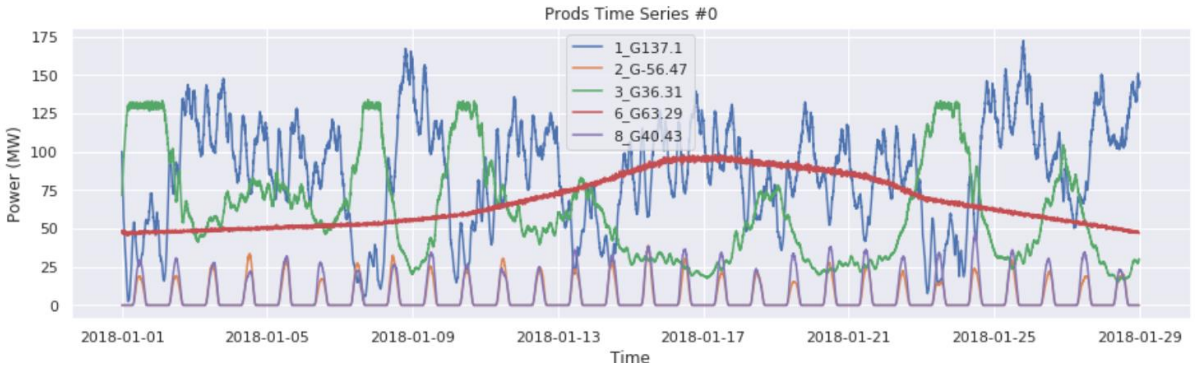

Figure 3 : Daily production of prods of the different substations

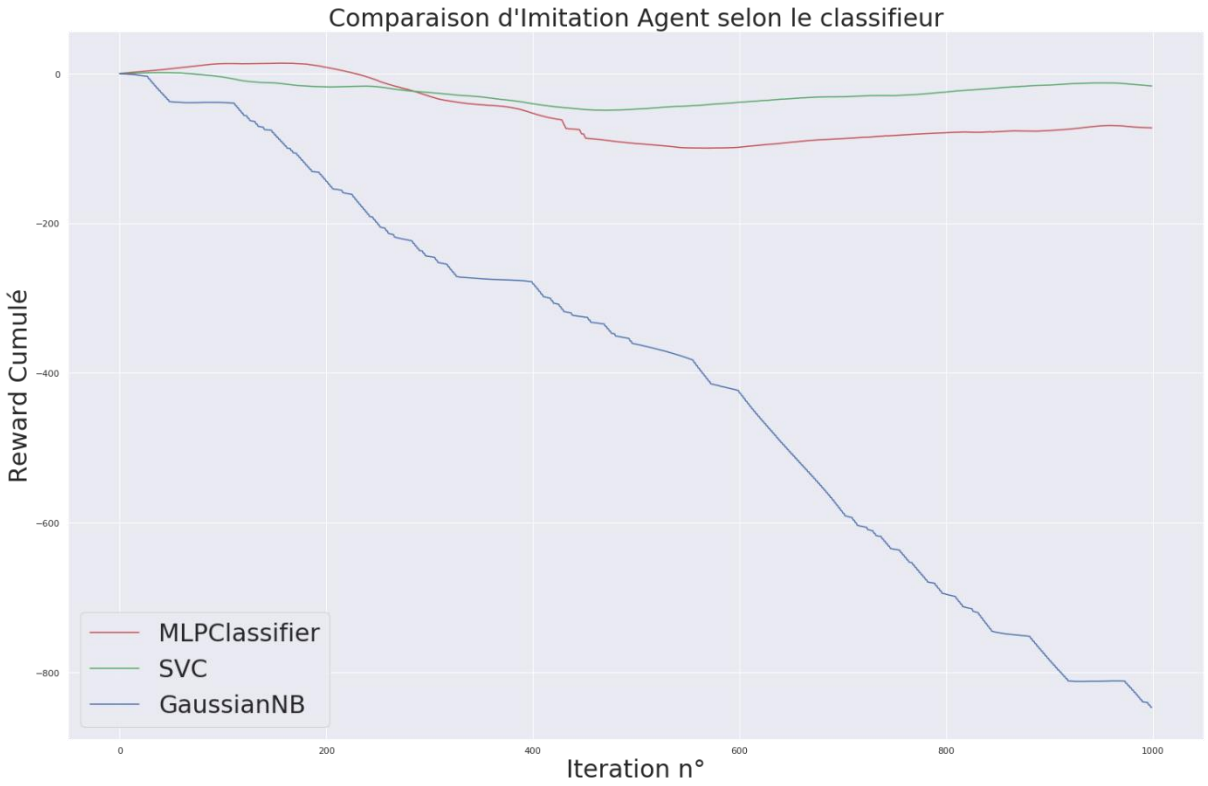

Figure 4 : Comparison of different classifiers through performance

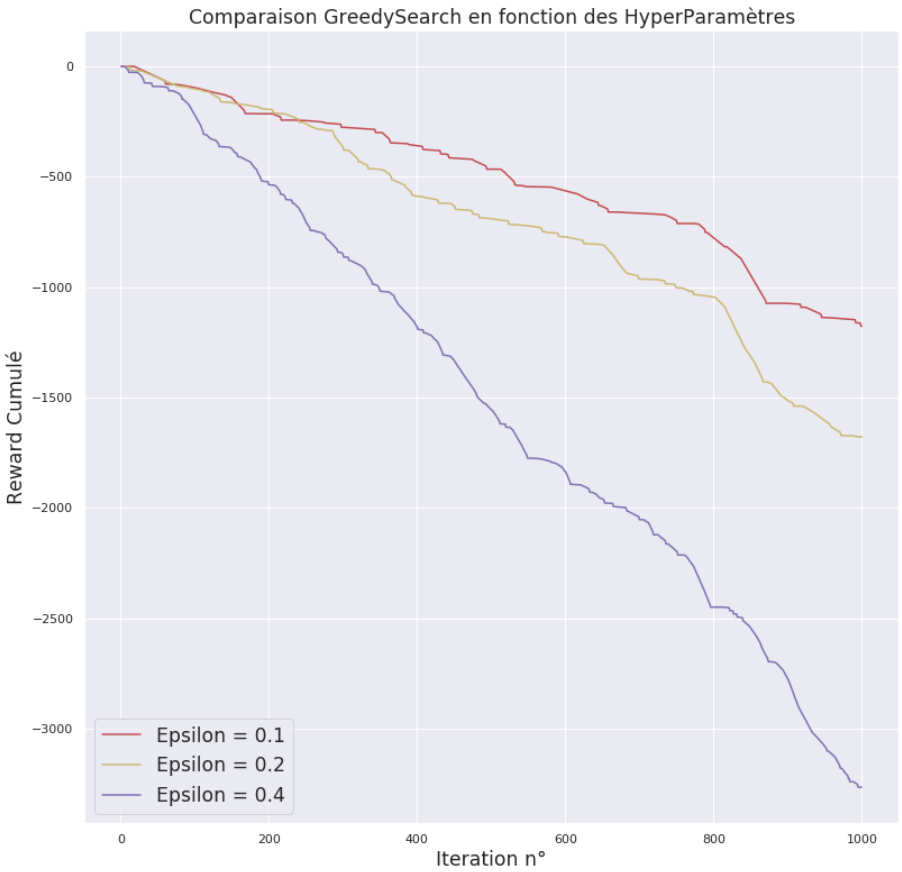

Figure 5 : Comparison of our GreedySearch agents with a varying epsilon

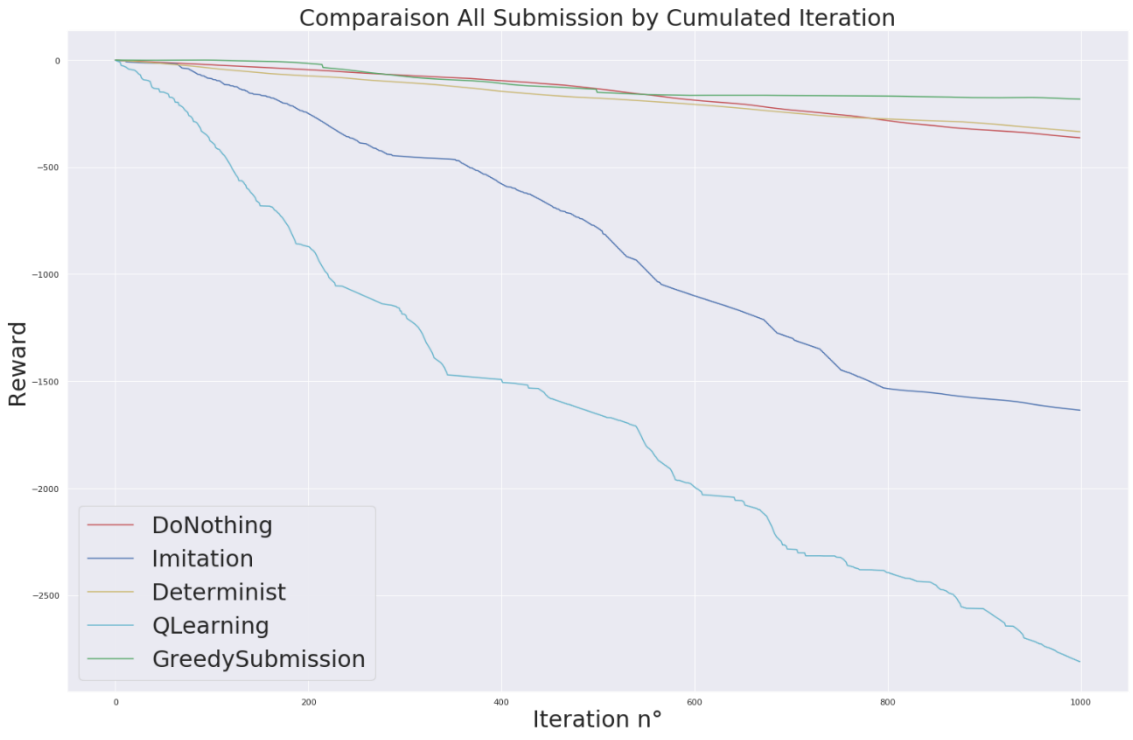

Figure 6 : Every agent built's cumulated reward

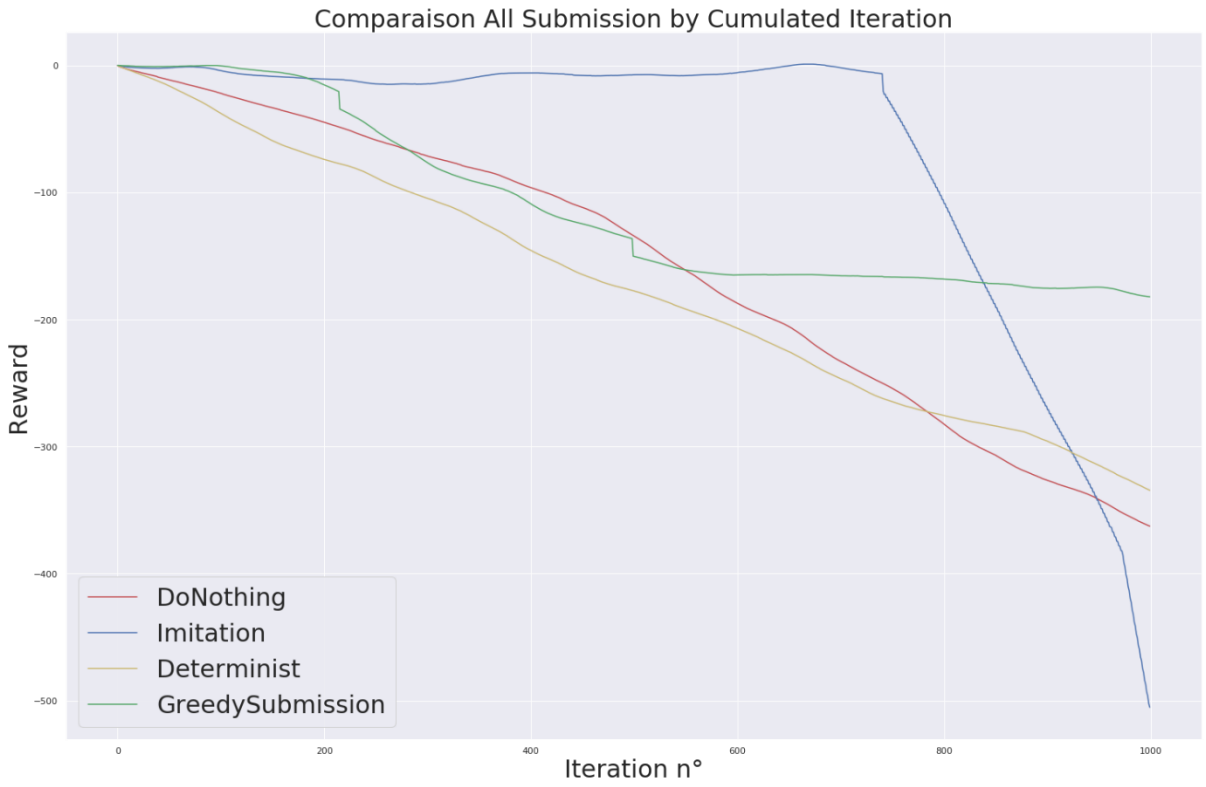

Figure 7 : Successful agents built's cumulated reward, on a rerun

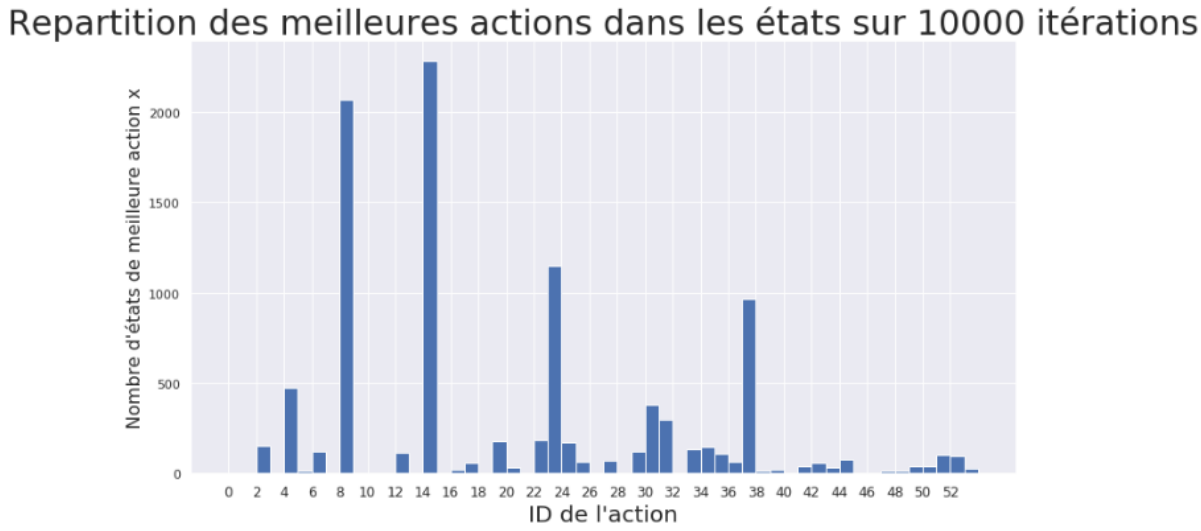

Figure 8 : Histogram of the actions taken by GreedySearch

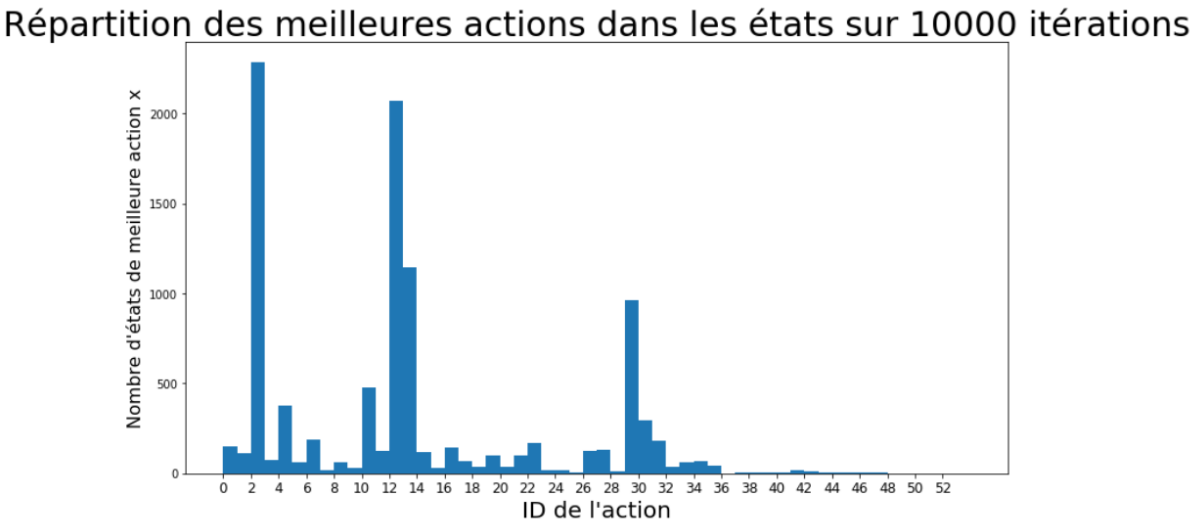

Figure 9 : Histogram of the actions taken by GreedySearch, on a re-run

Algorithms

```
Requierements : observation of our Grid

action <- doNothingAction
store capacity usage of each line n°i in linesCap[i]
randomely turn on a line which verifies linesCap[i] = 0
if lineCap[i] > 0.8:
    turn line n°i off
return action
```

Algorithm 1 : Pseudo-code of our first agent step function

```
Initialize s the current state
Initialize actionsAvailable, an array of actions with only 1 switch pressed,
or reconfigurations of the substations

bestReward <- -50 (-35 is the reward for a gameover, the worst outcome possible)
bestAction <- doNothingAction
for each action a in actionsAvailable:
    rewardByA <- computeReward(s,a)
    if bestReward < max(rewardByA,bestReward):
        bestAction <- a
        bestReward <- rewardByA
    end if
end for
return bestAction
```

Algorithm 2 : Pseudo-code of our GreedySearch algorithm

```
Needed method storeInfo(s,a) is a method which writes in 3 files;
one for states, one for actions, one for rewards, where each line of the
files corresponds to a triplet : state, best action from state, reward associated
Initialize s the initial state

a <- searchBestAction(s)
storeInfo(s,a)
epsilon takes a value between 0 and 1
if epsilon =< 0.1:
    randomely split a node
elif epsilon =< 0.2:
    randomely turn off a line
else:
    take the best action from s
```

Algorithm 3 : Pseudo-code of our ϵ-GreedySearch

Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:

Sates $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
procedure QLEARNING($\mathcal{X}$, A, R, T, α, γ)
    Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
    while Q is not converged **do**
        Start in state $s \in \mathcal{X}$
        while s is not terminal **do**
            Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow \arg\max_a Q(x, a)$)
            $a \leftarrow \pi(s)$
            $r \leftarrow R(s, a)$ ▷ Receive the reward
            $s' \leftarrow T(s, a)$ ▷ Receive the new state
            $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
            $s \leftarrow s'$
    return Q

Algorithm 4 : Pseudo-code of the Q-Learning algorithm

Bonus :

Table 1 : Final Rewards and Calculation Time for each agent

Method	Do nothing	Random line switch	Random node splitting	Greedy search	tree search
[Local] Final Reward	-11.2693	-123.58680	-156.71588	7.27904	-6.42783
Calculation Time (in s)	1.179	2.085	2.224	47.732	14.269

Metric :

This challenge's metric is the numerical value of rewards. As said in the introduction, they measure how good an agent is doing by looking at the environment and the state of the grid. Pypownet allows us to modify the way they are computed, but it's safer to use the default computing method to not skew our results any further. We've copied the full code at the root of our git repository, and we can see that the rewards are calculated in a pretty complex way. The most notable thing is that it doesn't only measures how close we are to fulfilling the needs of each loads, but it also looks at how complex the different actions are. Switching on/off a line or rearranging them has a cost, and it mirrors real life. That means that complex actions are punished by the metric. Another thing to note is that it also compares the current grid with the initial one, punishing the DoNothing agent that else would probably have a score of 0.

Cross Validation :

When you finally created a good algorithm with the good hyperparameter you want to know how it will perform in real life, but you have only your learning data. So, to answer this question, you are going to separate your learning data in two parts for example 60% of your data in the learning part and 40% in your test part. Your algorithm will train on the 60% and you will be able to test it on the 40% left. Thanks to this new result you will be able to readapt hopefully your hyperparameters or sadly change all the algorithm.

Overfitting :

As we have seen it in the last paragraph your algorithm may be good on the learning part and bad on the test one, but how can we explain this? This could be

counter-intuitive for you but finally when you think about it’s totally normal. For example, if you try to classify flowers and on your learning part you only have “normal” plants. You may find plants which have mutated in your test part and because you created perfect fit for normal plants, your algorithm will never be able to classify it in the good spot. This phenomenon is called overlearning, but the real thing is more overfitting of the equation on specific data that can’t represent all the possibility of real data.