Faculty of Applied Sciences

Optimal Decision Making For Complex Problems

INFO8003-1

Active Network Management (ANM) : Searching High-Quality Policies to Control a Complex Power Network

Authors :
Delcour Florian - s181063
Makedonsky Aliocha - s171197

Instructors :
Ernst Damien

Teaching assistants :
Aittahar Samy
Miftari Bardhyl

May 2022

Introduction

In this project, we consider the problem of controlling a complex power network known as *Active Network Management* (ANM). The environment ANM6-Easy takes part in the Gym-ANM [1] framework, which has been created in order to facilitate the design of RL environments that model ANM tasks in electricity distribution networks. In fact, Gym-ANM is a tool for designing RL models that operate complex power network while allowing to abstract from the complex dynamics of power system modelling.

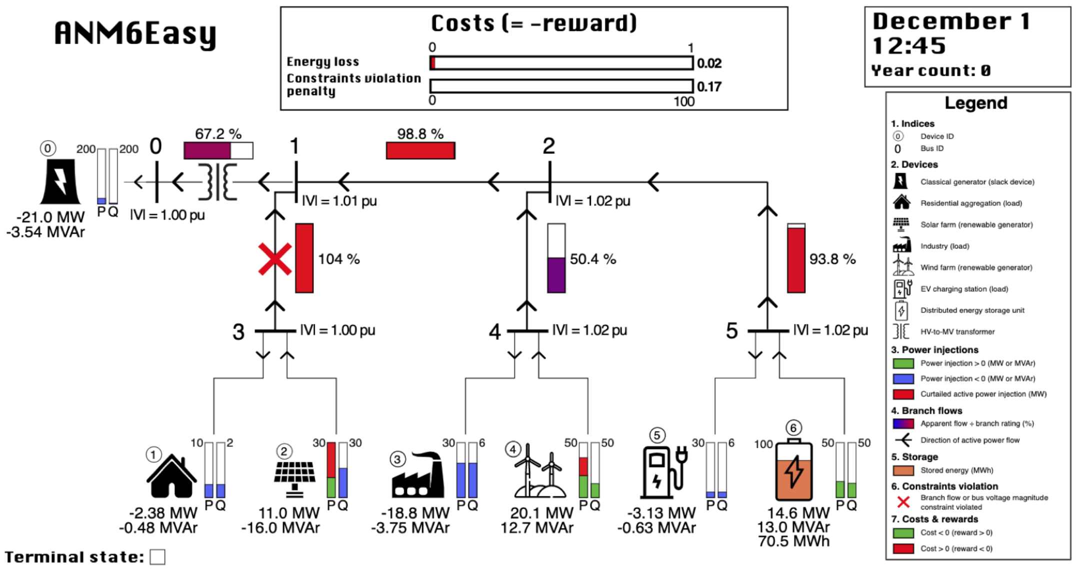


Figure 1: ANM6-Easy network

Figure 1 shows the ANM6-Easy network. It consists of 6 buses, with a high-voltage to low-voltage transformer, that are connected to the following devices : three passive loads, two renewable energy generators, one distributed energy storage unit (DES unit) and one fossil fuel generator used as slack generator. Generators, with the exception of the slack one, can only inject power into the grid while the loads can only withdraw power from it. Distributed energy storages (DES) can perform both : inject or withdraw.

The goal of this project is to design an agent, using *Reinforcement Learning* algorithms, which operates a power network by minimizing the operating costs while only performing secure operations for the distribution network in order to avoid grid collapse.

The rest of this document is organized as follows: Section 1 will formalize and characterize the environment and Section 2 will explained our implementation of two well-known policy gradient algorithms : *Reinforce* [2] and *Proximal Policy Optimization* (PPO) [3]. Eventually, we will evaluate their performance and compare them in Section 3.

Note : the following source code was provided to us as a reference implementation of the environment : `https://github.com/robinhenry/gym-anm`.

1 Domain

Provide the exhaustive list of characteristics of the domain as seen in the lectures.

1.1 Devices

ANM6-Easy network is composed of several devices annotated $\mathcal{D}$:

- ✦ Generators :
  1. Non-slack generators (renewable) : index 2 and 4 in Figure 1, those are the PV panels and the wind turbines
  2. Slack generator : index 0 in Figure 1, it's a fossil fuel generator (from the paper, even if it seems to be a power plant in Figure 1)

  The generators are annotated by g, and the set of generators by $G \subset \mathcal{D}$. They are only able to inject power into the grid, with the exception of g^{slack} which can either inject or withdraw power in order to balance the network (by the slack bus).
- ✦ Loads :
  1. House : index 1
  2. Factory : index 3
  3. Charging station : index 5

  They are only able to withdraw power from the grid.
- ✦ Distributed Energy Storage (DES unit) : index 6
  It can both inject and withdraw power from/into the network.

1.2 State space

The state space is $\mathcal{S} = \{s_t \mid \forall t \leq T\}$, where T is the time horizon and s_t consists in a set of state variables associated to every devices presented above, plus an auxiliary variable that, in the case of the ANM6-Easy environment, is the actual time of the day, measured every 15 minutes. We thus have

$$s_t = \left[\{P^{dev}_{d,t}\}_{d\in\mathcal{D}},\ \{Q^{dev}_{d,t}\}_{d\in\mathcal{D}},\ \{SoC_{d,t}\}_{d\in\mathcal{D}_{DES}},\ \{P^{max}_{g,t}\}_{g\in\mathcal{D}_G-\{g^{slack}\}},\ t_{day}\right]$$

with :

- $P^{dev}_{d,t}$ and $Q^{dev}_{d,t}$: active and reactive power injections of device $d \in \mathcal{D}$ into the grid. Since all devices are taken into account, we have 7 active powers and 7 reactive powers (indices : see Section 1.1).
- $SoC_{d,t}$: state of charge of DES unit $d \in \mathcal{D}_{DES}$.
- $P^{max}_{g,t}$: maximum production that generator $g \in \mathcal{D}_G - \{g^{slack}\}$ can produce.
- t_{day} : time of the day.

1.3 Action space

The action space is given by $\mathcal{U} = \{a_t \mid \forall t \leq T\}$ where a_t consists in a set of control variables associated to non-slack generator and DES unit devices presented in Section 1.1. At each timestep, the agent can only affect these control variables :

$$a_t = \left[\{a_{P_{g,t}}\}_{g\in\mathcal{D}_G-\{g^{slack}\}},\ \{a_{Q_{g,t}}\}_{g\in\mathcal{D}_G-\{g^{slack}\}},\ \{a_{P_{d,t}}\}_{d\in\mathcal{D}_{DES}},\ \{a_{Q_{d,t}}\}_{d\in\mathcal{D}_{DES}}\right]$$

with :

- $a_{P_{g,t}}$: upper limit on the active power injection from generator $g \in \mathcal{D}_G - \{g^{slack}\}$ (curtailment if $g \in \mathcal{D}_{DER} = \mathcal{D}_G - \{g^{slack}\}$: set-point chosen by the agent for classical generators, slack generator is excluded since it will always fulfill the needs of the network if it lacks of generation)
- $a_{Q_{g,t}}$: reactive power injection from each generator $g \in \mathcal{D}_G - \{g^{slack}\}$. Once again, slack generator cannot be controlled by the agent since it will fulfill the needs.
- $a_{P_{d,t}}$: active power injection from each DES unit $d \in \mathcal{D}_{DES}$
- $a_{Q_{d,t}}$: reactive power injection from each DES unit $d \in \mathcal{D}_{DES}$

The boundaries on the continuous action space are the following :

☆ $P_{PV} \in [0\ ;30]$ and $P_{wind} \in [0\ ;50]$

☆ $Q_{PV} \in [-30\ ;30]$ and $Q_{wind} \in [-50\ ;50]$

☆ $P_{DES} \in [-30\ ;30]$ and $Q_{DES} \in [-50\ ;50]$

The control variables are restricted to finite ranges $[P_{inf}, P_{sup}]$ or $[Q_{inf}, Q_{sup}]$ because electrical devices cannot inject or withdraw infinite power. Moreover, for generators and DES units, they have current injections limits, such that they cannot operate at full capacity for both active and reactive power. Lastly, the DES units power injections depend on their current storage level.

1.4 Dynamics

The dynamics of such a system are complex and were not asked in the statement. However, it is explained in appendix of R. Henry and D. Ernst paper [1].

1.5 Reward signal

The reward signal is given by :

$$r_t = \begin{cases} clip\left(-r_{clip}, -(\Delta E_{t:t+1} + \lambda\phi(s_{t+1})), r_{clip}\right) & \text{if } s_t \text{ terminal} \\ -\dfrac{r_{clip}}{1-\gamma} & \text{if } s_t \text{ not terminal but } s_{t+1} \text{ terminal} \\ 0 & \text{else} \end{cases}$$

with :

- $\Delta E_{t:t+1}$: the total energy loss from t to $t+1$
- $\phi(s_{t+1})$: a penalty associated to the violation of operation constraints
- λ : a weighting parameter, set at 1000
- $r_{clip} > 0$: keeps a reward in the range $[-r_{clip}; r_{clip}]$. Here $r_{clip} = 100$

The reward is clipped to avoid too big rewards that would make the problem really unstable. The system is in a terminal state if the grid has collapsed, thus the simulation stops when it's the case.

1.6 Observation space

In the case of `ANM6-Easy` network, the environment is fully observable such that at each timestep, the observations correspond to the current state of the environment.

1.7 Others

Some other relevant information :

- ➤ The environment is deterministic, there is no randomness.
- ➤ The discount factor γ is fixed to 0.995.
- ➤ The timestep Δt considered is $0.25h$ = 15 minutes.
- ➤ We will use a time horizon of 4800 (50 days) timesteps to train our network. However, in the end, the agent should be able to keep the network alive infinitely.

2 Policy Gradient Techniques

In the previous assignments, almost everything we have achieved has been through Q-learning methods. While Q-learning aims to predict the reward of a specific action performed in a specific state, policy gradient methods predict the action itself.

The main idea underlying policy gradients is to reinforce good actions by increasing the probabilities of actions that lead to higher returns and decreasing the probabilities of actions that lead to lower returns until the optimal policy is reached.

2.1 Policy Gradient Algorithm : *REINFORCE*

Reinforce is a policy gradient algorithm, which means that the neural network we will use directly tries to learn an optimal policy by outputting action instead of learning to output good estimates of the Q-function, as seen in previous assignments. As it outputs an action, policy gradient is really useful for continuous action space, which is the case in the current project.

Reinforce is an example of on-policy algorithm, which means that we use the policy that we are learning to generate the trajectories, which are then used to optimize the weights of our network. This as the consequence to guide the exploration in the region we trust to be promising, instead of doing random exploration.

Algorithm 1: REINFORCE

```
Select learning rate $0 < \alpha \leq 1$;
Set discount factor $\gamma$;
Choose number of epochs $N$;
Choose length of episode $T$;
for $N$ epochs do
    Generate an episode $s_0, a_0, r_0, ..., s_t, a_t, r_t$ following policy $\pi(a \mid s, \theta)$;
    for $t$ from $T-1$ to $0$ do
        $G_t \leftarrow \sum_{k=t}^{T-1} \gamma^{k-t} \cdot r_k$;
    end for
    $\mathcal{L}(\theta) \leftarrow -\frac{1}{T} \sum_{t=0}^{T-1} G_t \cdot \ln \pi_\theta(a_t|s_t, \theta)$;
    $\theta \leftarrow \theta + \alpha \nabla \mathcal{L}(\theta)$;
end for
```

Algorithm 1 shows the pseudo-code of our implementation where we set $\alpha = 0.001$, $N = 100$, $T = 4800$.

As we want to increase the expected cumulative return, *Reinforce* performs gradient ascent. However, one can see that there is a minus sign in the loss computation which allows for gradient descent and is explained in this article [4]. Deep Learning algorithms are more designed for gradient descent.

2.1.1 Action space

As previously mentioned, the action space is continuous. *Reinforce* has the advantage to easily manage continuous action space. However, we also implemented a version where we discretized the action space to make some comparison. The action space has been discretized with 500 steps between the action variables boundaries.

2.1.2 Architecture

1. Network for continuous action space
   - ☐ Input size : 18 = number of state variables
   - ☐ Output size : 6 = number of action variables
   - ☐ 3 linear layers
   - ☐ Hidden layer size : 512
   - ☐ Activation : ReLU for hidden layers and none for output layer
   - ☐ Adam optimizer (learning rate = 0.001) : not too high to avoid divergence, but not too small to allow the loss to improve.

This is the network for which we will compare its performance later. We also tried several different configurations : changing hidden layer size, adding activation function at output (sigmoid, tanh), and so on, but these gave poorer results. Without activation function to the output, the agent will output 6 actions variables that are unconstrained. We must therefore clip them in the range of the action variables boundaries, defined in Section 1.3, before sending it to ANM6-Easy environment. And we penalize the network when it outputs action variables outside their boundaries.

2. Network for discretized action space
   - ☐ Input size : 18
   - ☐ Output size : 6×500 because action space discretized with 500 steps
   - ☐ 2 linear layers
   - ☐ Hidden layer size : 512
   - ☐ Activation : ReLU for hidden layer and softmax for output layer to obtain probabilities
   - ☐ Adam optimizer (learning rate = 0.001)

2.1.3 Trajectory generation

1. Continuous action space :

   To generate a trajectory, we draw an action from a multivariate normal distribution where the mean is the 6 outputs from the network and the covariance matrix is a diagonal matrix with 6 diagonal elements (standard deviations). These standard deviations are also learned by the model. Then, we make a step in the environment by sampling an action from the distribution, and we collect the next state and the associated reward. We do that until the grid collapses, or the length of the trajectory is equal to 50 days (50*96 timesteps).

2. Discrete action space :

   For the discretized action space model, it's quite similar but the network outputs 500 probabilities for each control variable. We then sample an action from these 6 categorical distributions. The rest is similar as for continuous action space.

2.1.4 Policy optimization

This section is similar for both continuous and discretized action space.

To optimize the policy, we need to compute the discounted rewards with the rewards collected during the episode, along with the log probabilities of the actions collected. Then we take as loss

$$\mathcal{L}(\theta) = -\frac{1}{T}\sum_{t=0}^{T-1} G_t \cdot \ln\ \pi_\theta(a_t|s_t, \theta)$$

that we try to minimize, which is equivalent to maximize the opposite of the loss :

$$\mathcal{L}(\theta) = \frac{1}{T}\sum_{t=0}^{T-1} G_t \cdot \ln\ \pi_\theta(a_t|s_t, \theta)$$

That's why we talked about gradient ascent. The agent has control on the action variables and thus on the log probabilities and not on the expected rewards G_t. Since the probabilities are included in $[0\ ;\ 1]$, their log are in the interval $]-\infty,\ 0]$, thus always negative (or equal to 0). Therefore, we have two possible cases (the graph of the logarithm is shown in Figure 2 to understand correctly what we present) :

- $G_t < 0$: the loss is thus negative. Since we try to minimize it, the agent will increase the log probability in absolute value, which is done by minimizing the probability of the associated action variables. It's a pleasant behaviour since having negative expected rewards means that the actions taken were bad (we always try to maximize the expected rewards). Thus we minimize the probability of taking those bad actions.
- $G_t > 0$: the loss is thus positive. To minimize it, we must decrease the log probability in absolute value, which is equivalent to increasing the probability of taking the corresponding action variables. It's also a pleasant behaviour since the corresponding action variables provide good rewards since the expected rewards are positive.

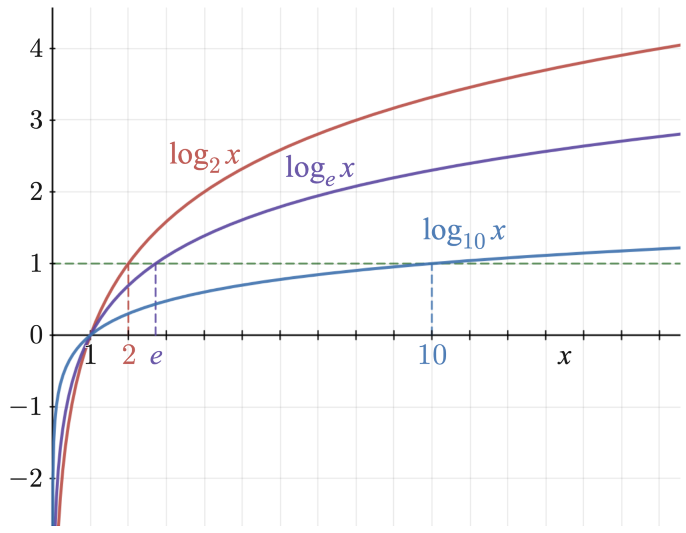


Figure 2: Plot of the logarithm function (whatever the basis)

2.1.5 Action variables boundaries

The model could output any action, but the environment only accepts action variables in the boundaries described in Section 1.3. For discretized action space, the agent is forced to output probabilities associated to action variables that are in the range of their boundaries. For continuous action space, as already introduced, we tried several configurations. First we put a sigmoid activate layer at the end of the neural network. The agent then outputs action variables in [0,1] and we only need to unscale these before passing it through the ANM6-Easy environment. We also tried a tanh activation that ouputs action variables in [-1,1]. However, these gave poorer results than our last method.

It consists to penalize action variables that are outside their boundaries, in order to make the agent learn these boundaries. In fact, if one or several action variables are outside of their boundaries, we attribute a reward $r = -r_{clip} * nb\_action\_var\_outside$.

After attributing penalized reward, we simply put the actions to their closest bound (if an action is smaller than the lower bound we put it to the lower bound, if bigger than the upper bound we put it to the upper bound) before making the step in the environment with those. That's not a clean practice because the network isn't aware that its output is way out of bounds, but the penalized reward seems to works quite well.

Figure 3 and 4 shows the mean and standard deviation of the expected return over 5 episodes, with activation sigmoid and tanh. The results are disappointing compared to the one obtained with penalized reward and no activation (Figure 11).

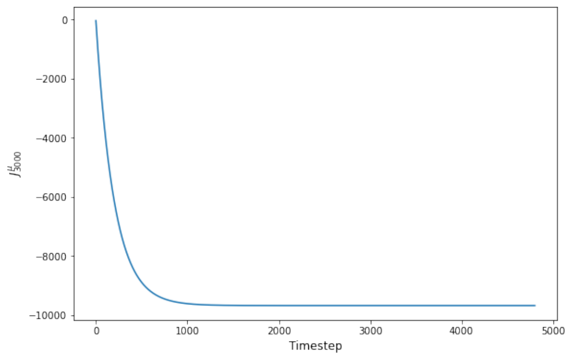


Figure 3: Sigmoid activation, expected return, over 5 episodes (T=3000)

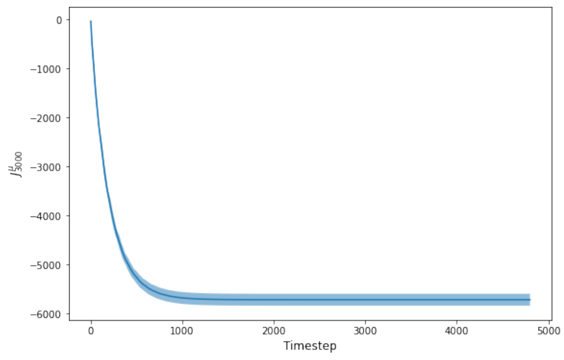


Figure 4: Tanh activation, expected return, over 5 episodes (T=3000)

2.2 Proximal Policy Optimization (PPO)

The proximal policy optimization algorithm is an actor critic and policy gradient method which then combines their advantages.
The principle is the following : we have two neural networks, one is the actor which will learn the optimal continuous action variables to take given a state, thanks to policy gradient approach with gradient descent. The other one is the critic that will learn a value function to be able to criticize the action variables took by the actor given a state.

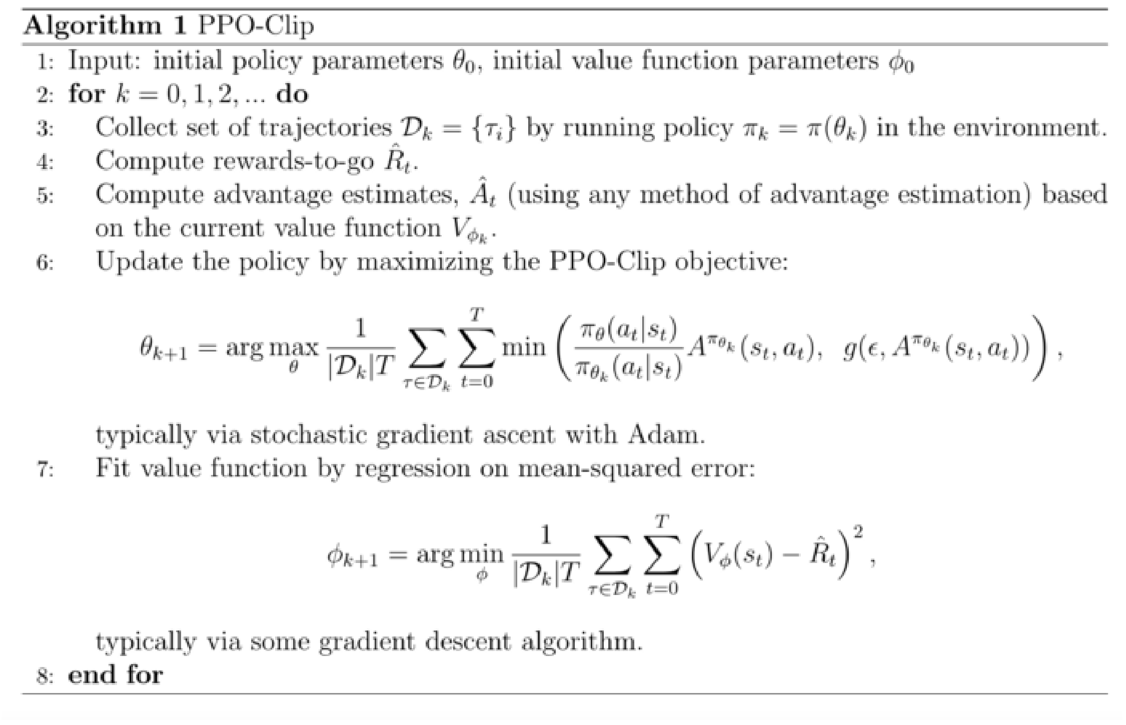
2.2.1 Architecture

The architecture used for those two networks consists in :

- ☐ Input size : 18
- ☐ Output size : 6 for actor network and 1 for critic network
- ☐ 3 linear layers
- ☐ Hidden layer size : 64
- ☐ Activation : tanh for input and hidden layers
- ☐ Adam optimizer (learning rate = 0.002)

2.2.2 Algorithm

The following implementation is inspired from [5].
In Figure 5 is the algorithm used for PPO.



```
Algorithm 1 PPO-Clip
1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: for $k = 0, 1, 2, ...$ do
3:     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4:     Compute rewards-to-go $\hat{R}_t$.
5:     Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
       on the current value function $V_{\phi_k}$.
6:     Update the policy by maximizing the PPO-Clip objective:
           $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t))\right),$
       typically via stochastic gradient ascent with Adam.
7:     Fit value function by regression on mean-squared error:
           $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$
       typically via some gradient descent algorithm.
8: end for
```

Figure 5: PPO algorithm

The initial parameters θ_0 and ϕ_0 refer, respectively, to the weights of the neural network for the actor and for the critic. Then $k = 0, 1, 2..., n$ represents a loop that will run n times, instead we will run the loop until having trained the models over 10^6 timesteps.

2.2.3 Trajectory generation

The next step is to generate a trajectory using the policy given by the actor network. The trajectory will consist in a batch of 3 episodes. Those episodes are computed on at most 50 days, so $50 * 96 = 4800$ timesteps, but if the grid collapses the episode stops instantly. Neither the actor nor the critic networks are updated during this collection of events. To select an action at each timestep we give the observations to the actor network, that outputs 6 values, one for each action, and we draw 6 actions from 6 normal distributions with as means the values outputed by the network and 0.5 as variance. But we had to clip the actions given to the closest bounds otherwise there were problems with the environment (as explained for REINFORCE, but without the penalty for out-of-bounds actions). We tried to do that in the neural network through a scaled sigmoid layer at the end but the results were not satisfying. Once we have collected a batch of 3 episodes, we can compute the rewards to go from all of our rewards, which are the discounted rewards basically. Then the advantage estimates, computed by substracting the V functions, outputed by the critic network when we feed the different states collected to it, from the rewards to go that we just computed. Then we subtract its mean to it and we divide the whole by its standard deviation.

2.2.4 Optimization of the policy and the value function

The next step is to update the weights of the actor network, thus our policy. To do so we will do n epochs, we choose $n = 5$, during which we will compute the V function with our critic network (that is updated at each epoch), the log probabilities of the actions computed with our actor network (also updated at each epoch), compute the losses for both the networks then updates those with backpropagation, and do that for n epochs.
The loss for the critic network is simple a mean-squared error between the value function outputed and the reward to go, but the loss for the actor network is a bit more complex, it's called *Clip Surrogate Objective*. Its effect is shown in the Figure 6, where the A shows the advantage estimates described above.

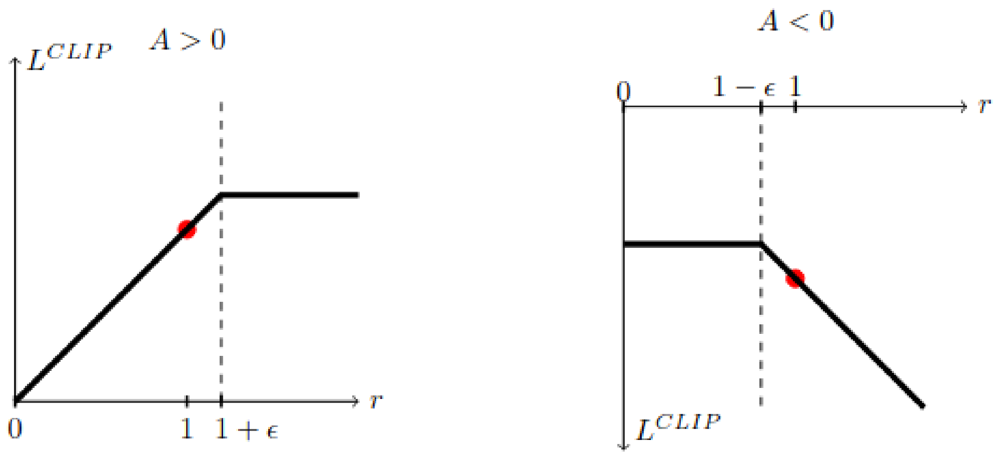


Figure 6: Clip surrogate objective effect

It keeps the ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$ (new policy over the previous policy) shown in Figure 5 between $[1-\epsilon, 1+\epsilon]$ to avoid big changes. ϵ is the clipping parameter and is generally set to 0.2. Then we multiply this clipped ratio by the advantage estimates and we get the clipped objective. Finally we take the min between the clipped objective and the unclipped one, thus we get a lower (pessimistic) bound on the unclipped objective. It means that when the change in the probability ratio would improve the objective, we ignore it, and we only include it when it makes the objective worse.

And we repeat this algorithm until a total of N (1 million for us) timesteps have been collected and used to train.

3 Results

In this section, we present our results by showing the training loss of each model, comparing the policies in terms of *expected discounted cumulative reward* obtained at the end of each episode, for final trained model and also over training.

Figure 7, 8 and 9 show the evolution of the loss of each algorithm during their training. For PPO algorithm, we only show the loss for the actor network since it's the network that will determine our policy.

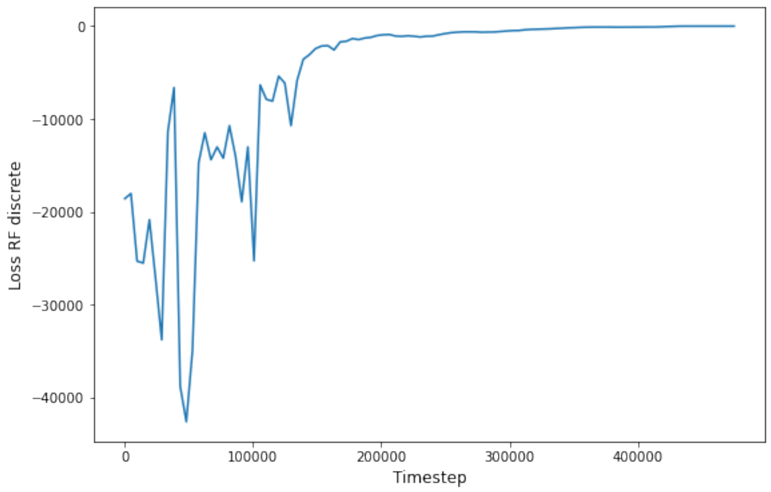


Figure 7: Loss for REINFORCE with discretized action space

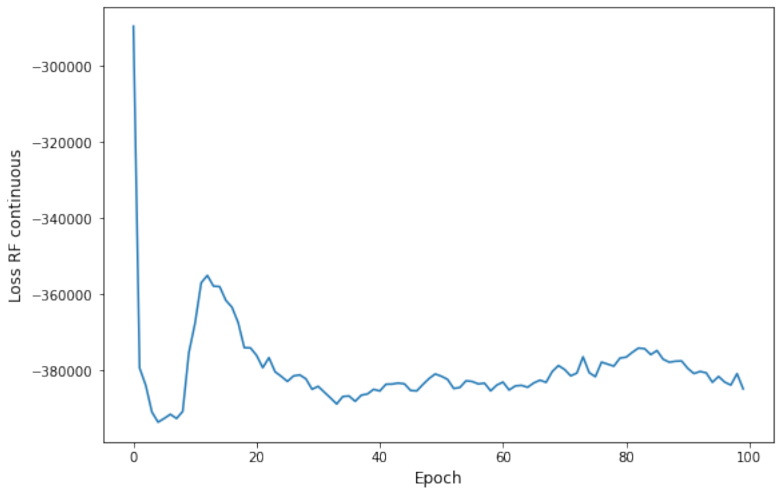


Figure 8: Loss for REINFORCE with continuous action space

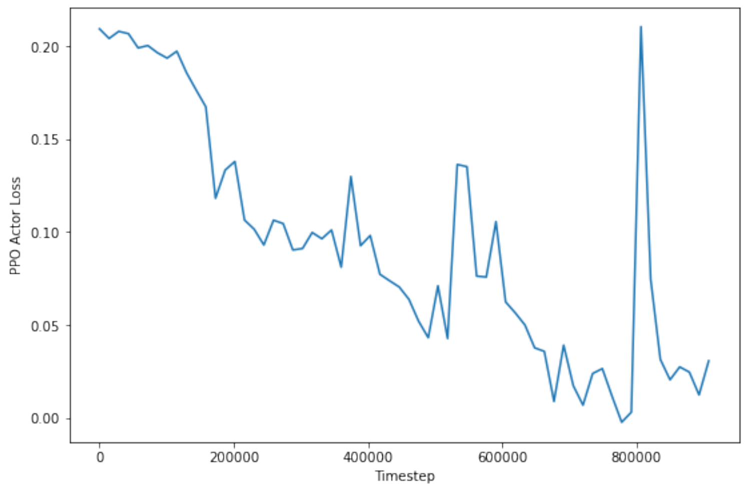


Figure 9: Loss for PPO (actor net.)

As explained in Section 2.1, for REINFORCE we perform gradient ascent thus we maximize the loss, whereas for PPO, we do gradient descent and thus the loss is minimized. The evolution of the loss for REINFORCE with continuous action space seems a bit strange with this reasoning. However, we got really bad results so we don't pay to much attention to this loss. The continuous REINFORCE doesn't really learn, which will be shown later when the expected return is worse after training than before.

To evaluate the different models, we trained them over 50 days which correspond to 480 000 timesteps. We would have liked to train them more but it was really long to train so we ran out of time. We used the final trained models to compute their mean and standard deviations of expected returns over 5 episodes, each one of T=3000 timesteps and compare them. Figure 10, 11 and 12 show the expected return with the algorithms fully trained. The mean over the 5 episodes is the curve in dark blue, and the standard deviation is the light blue area.

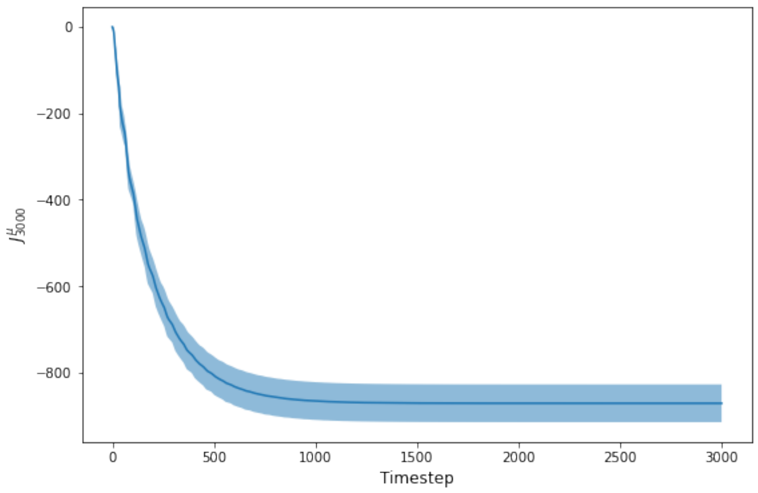


Figure 10: Expected return over 5 episodes (T=3000) for discrete REINFORCE with full training

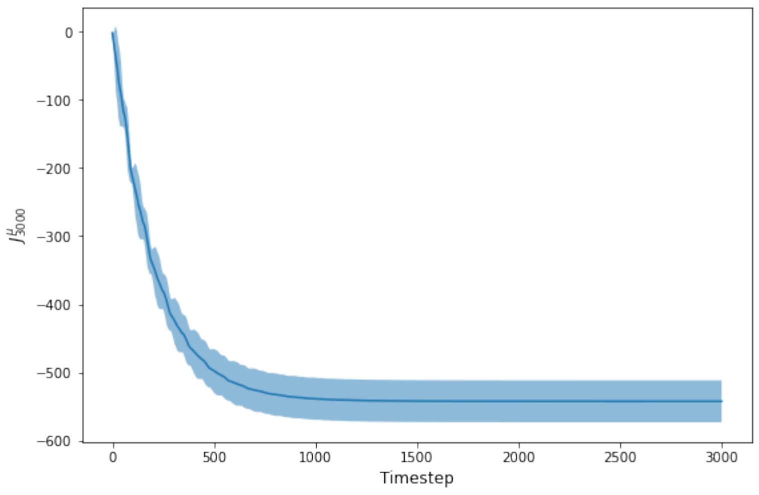


Figure 11: Expected return over 5 episodes (T=3000) for continuous REINFORCE with full training

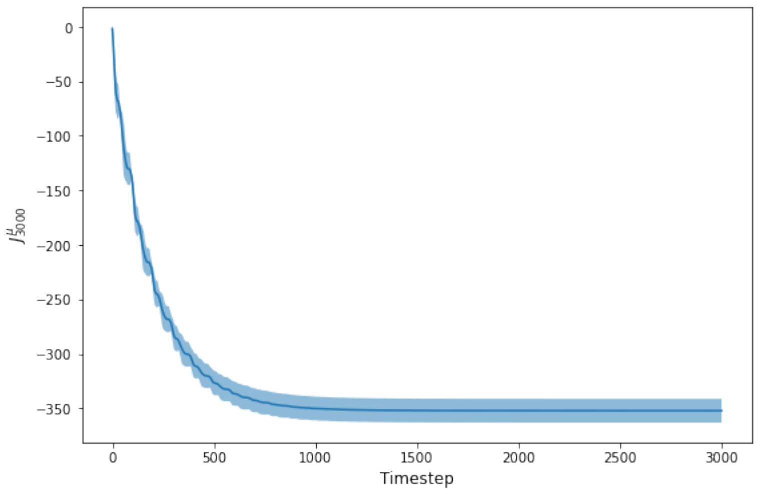


Figure 12: Expected return over 5 episodes (T=3000) for PPO with full training

We observe that PPO is the one that achieves the best expected return with a convergence around -350, while REINFORCE converges as quickly but with a much smaller expected return (-850 for the discrete version, -550 for the continuous version, which is better than discrete version but worse than PPO). It is known that REINFORCE learns really slowly, which may explain the difference in expected return. Maybe if we had trained them over 3 million timesteps we would have observed better results with REINFORCE.

Now we observe the mean and standard deviation of the expected return still over 5 episodes (T=3000 timesteps), but for different stages of training for the different algorithms.

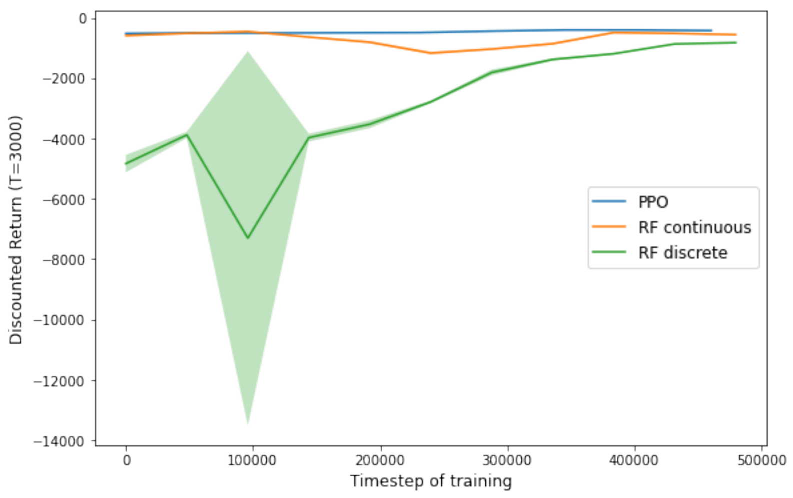


Figure 13: Expected return for the 3 algorithms, for different training stages

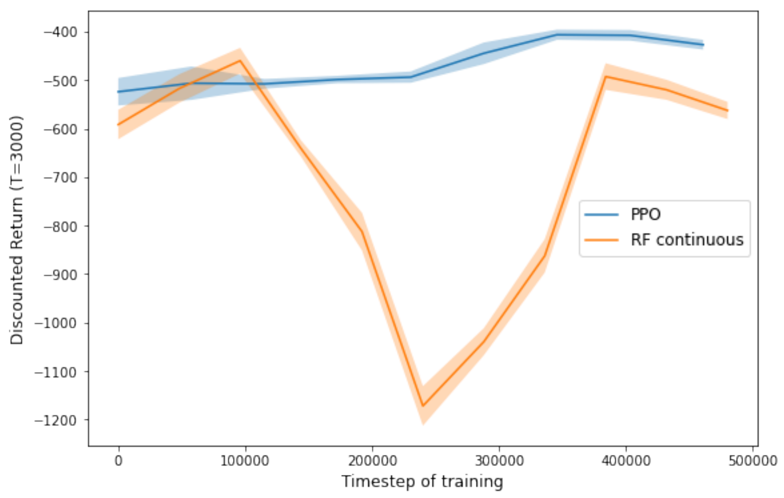


Figure 14: Expected return for continuous REINFORCE and PPO, for different training stages

The performance of discrete REINFORCE are way worse than the ones of the continuous REINFORCE and PPO in the beginning, but after the full training, the gap becomes smaller (but still present as shown in the graphs before). Continuous REINFORCE was really close to PPO with few training but its performance quickly decreases and PPO becomes unquestionably better after 150 000 timesteps of training.

We also tried to compare how much time the algorithms would survive in the environment. After 3000 consecutive days (= 288 000 consecutive timesteps), the grid hadn't still collapsed. Thus we assumed that the algorithms fully trained were good enough to survive the environment, which is the main goal of this project.

4 Possible improvements

As explained, our method to keep the action variables outputted by the continuous REINFORCE and PPO algorithms isn't clean at all, we should investigate for a method that would make the networks (actor network for PPO) output directly an action in its corresponding bounds.

We should also train much more the algorithms, we lacked of computing power which made the training very slow and that prevented us from trying several parameters combinations that we could have tried. Making more training to allow the algorithms to see more states (and since its a continuous environment, seeing more state is really important since there is an infinity of possible states so any more experience taken is profitable). Moreover, REINFORCE is known as a slow learning algorithm, but PPO would also profit from a longer training.

References

[1] Robin Henry and Damien Ernst. Gym-anm: Reinforcement learning environments for active network management tasks in electricity distribution systems. *Energy and AI*, 5:100092, 2021.

[2] R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[3] P. Dhariwal A. Radford O. Klimov J. Schulman, F. Wolski. Proximal policy optimization algorithms. 2017.

[4] Adrien Lucas Ecoffet. An intuitive explanation of policy gradient. 2018.

[5] Eric Yang Yu. Coding ppo from scratch with pytorch. 2020.