RLD

Reinforcement Learning

Alexander Baumann
Yunfei ZHAO

December 28, 2021

In this report we aim at demonstrating some basic elements and algorithms of reinforcement learning, with some implementations and demonstrations of questions from TMEs.

All **source code** of this report can be found on github by the following link: `https://github.com/YunfeiZHAO/RLD`, The authors of source code are Alexander Baumann and Yunfei ZHAO. The project is based on given code[1] from M.Sylvain Lamprier.

Contents

List of Figures

Chapter 1

TME1: Bandits Problems

1.1 Introduction

Bandits problems is a n-action one state MDP (Markov Decision Process) problem and we suppose that the actions we take will not affect world states which means current state is not depend on actions we have taken. We will make decision online and get reward immediately. Further more, we have little information about environment, and obviously, we do not know the reward we will get by taking an action. So we need to exploit information from experience and in the same time do the exploration.

The problem we are going to solve for this TME is that we have 5000 articles (samples) with theirs five dimensional description features and the click-through rate of advertisements from 10 advertisers and we suppose that we only know the clic-through rate after choosing an advertiser for an article. So we can only get information of clic-through rate for articles we attributed an advertisers. So advertisers in this case is our bandits and we suppose the article has no influence of advertisers.

1.2 Algorithms

We will test five algorithms for this problem. They are respectively:

- **Random Strategy**

  We will randomly attribute advertiser to articles.

- **Statically Best Strategy**

  We cheat here by using the mean reward of all advertisers calculated from 5000 samples which we would not have had access. And we always select the advertiser with the highest mean clic-through rate.

- **Optimal Strategy**

  We cheat here too by using the best reward of advertisers for one article which we would not have had access, because we do not know the reward before attribute an advertiser for the article.

- **UCB**

- **LinUCB**

1.2.1 UCB: The Upper Confidence Bound (UCB) Bandit Strategy

From here, we stop cheating, but what should we do? If we always choose "bandit" with max empirical reward, we will fall into sub-optimal option, for example, we choose bandit one for the first time, then we will always choose this one as mean reward for others is zero. If we use ϵ-greedy to choose bandits randomly time to time, the result will strongly depend on the value of ϵ. So we want to find a strategy that bound the regret:

$$\mathbb{E}(\rho_n) = n \times \mu^* - \mathbb{E}(\sum_{t=1}^{n} \mu_{\pi_t})$$

by O(log(n)), where n is the number of actions we took and μ^* is the experience of the best bandits and ρ_n:

$$\rho_n = \sum_{t=1}^{n} g(w_{i^*,t}) - \sum_{t=1}^{n} g(w_{\pi_t,t})$$

where g is the reward function and $w_{i,t}$ the result play with bandit i at time t.

For UCB, We following the strategy that:

$$\begin{aligned} \pi_t &= \underset{i}{argmax}\, B_{t,T_i(t-1)}(i) \\ B_{t,s}(i) &= \hat{\mu}_{i,s} + \sqrt{\frac{2log(t)}{s}} \end{aligned} \tag{1.1}$$

with $T_i(t)$ the number of time bandit i is played at time t. Intuitively, It suppose that bandits which have been played less may have bigger potential and it has been proved by Chernoff-Hoeffding theory with this algorithm:

$$\mathbb{E}(\rho_n) \leq K\frac{8log(n)}{\Delta^*} + K\Delta^*(1 + \frac{\pi^2}{3})$$

1.2.2 LinUCB

LinUCB is UCB with taking into account of individual context. In our case, individual context is the first 5 dimensional features. So we will use ridge regression to estimate the reward we are supposed to get together with UCB algorithm.

1.3 Experiment

From Figure 1.1, we can observe that for random algorithm and other two cheating algorithms, we have a linear accumulative reward curve. Of course that the two cheating algorithms achieved very good results. As for UCB and LinUCB they both converge quickly to have a pent of the statistical best strategy which meastend to chose the best statistical bandit exponentially often than other bandits. Further more, Lin UCB has better performance by considering context variables.

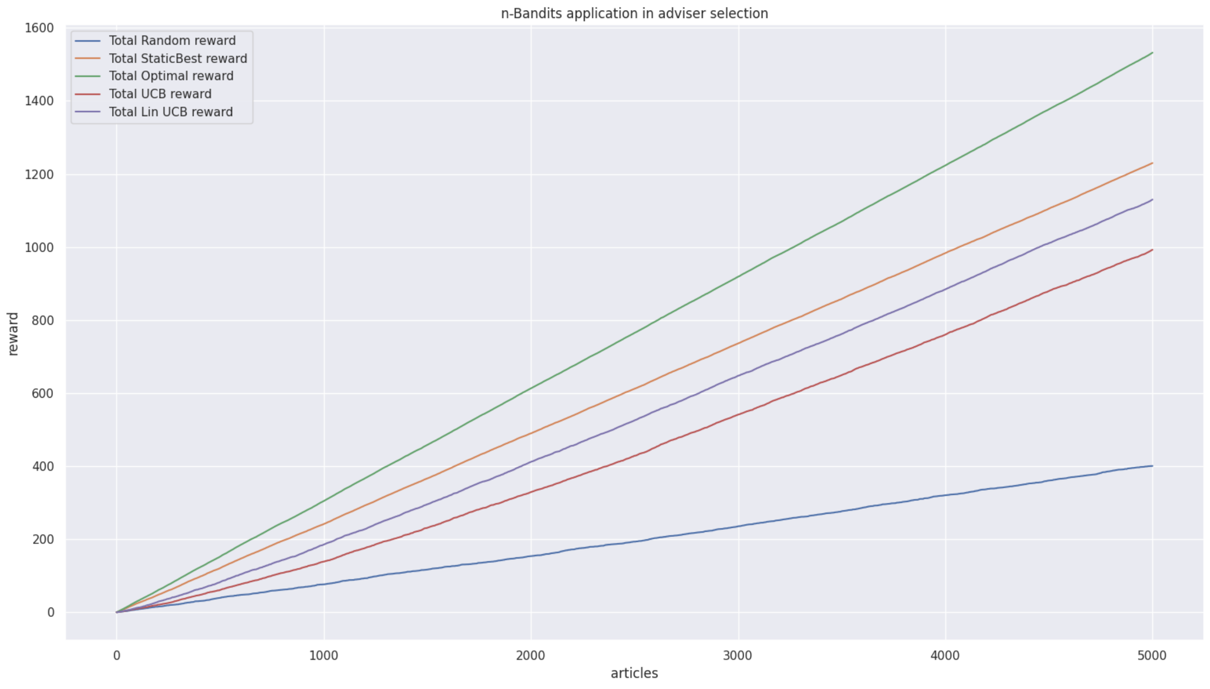


Figure 1.1: Reward curve of five different strategy

Chapter 2

TME2: Dynamic Programming

2.1 Introduction

In this part, we study value based reinforcement learning. We suppose the problems we deal with, for now, have discrete states, actions and their dimensions are relatively small. Besides, we possess MDP model of the environment. So in this part, we discuss, value-based and model-based offline solution by dynamic programming.

The main idea of this part is:

- Value function

  According to a policy π, value function $V^\pi(s)$ estimate how good to be in a state s.

$$V^\pi(s_t) = \mathbb{E}_\pi[R_t|s_t = s] = \mathbb{E}_\pi[\sum_{i=0}^{i=+\infty} \gamma^i r_{t+i}|s_t = s] \tag{2.1}$$

- Bellman equation

  We use bellman equation to make function V converge and it can give a fairly good estimation as respect to the experience. And Bellman equation refer to dynamic programming equation associated with discrete-time optimization problems. As we estimate $V(s_t)$ by $V(s_{t+1})$

$$\begin{aligned} V^\pi(s_t) &= \mathbb{E}[r_t] + \mathbb{E}[\gamma \sum_{i=1}^{i=+\infty} \gamma^{i-1} r_{t+i}|s_t = s] \\ V^\pi(s) &= \sum_{a\in A(s)} \pi(a,s) \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma V^\pi(s')) \end{aligned} \tag{2.2}$$

In our case, the value function is construct of a table of value, each value correspond to a state, and with a given policy, we will iterate this table thanks to Bellman equation until it converge to get the value function for our policy.

During the TME, we tried two types of algorithms in gym's grad **gridworld** environment. This environment is a MPD environment, each state correspond to a two dimensional matrix which encode objects for each position and we are provided with the transition matrix from one state to another and the rewards that our agent move to an object can be defined by us.

2.2 Algorithms

2.2.1 Policy Iteration

Policy iteration performs policy evaluation by value function and policy improvement alternatively. We define that a policy π is better than a policy π' when we have

$$\pi \geq \pi' \leftrightarrow \forall s : V^{\pi}(s) \geq V^{\pi'}(s)$$

For each policy, we evaluate the corresponding value function by:

$$V^{\pi}(s) = \sum_{a \in A(s)} \pi(a, s) \sum_{s'} P(s, a, s')(R(s, a, s') + \gamma V^{\pi}(s'))$$

So after getting the value function, we choose the best action according to value function and MPD as the action we will take for the current state to generate a new policy. We stop the process until we get the same policy after an update.

2.2.2 Value Iteration

Policy iteration algorithms converge certainly to a stationary optimal policy in a finite time. But for each iteration, we evaluate the whole policy this can be very expensive on big map. According to Bellman equation we found that:

$$\begin{aligned} V^*(s) &= max_{\pi} V^{\pi}(s) \\ &= max_a \sum_{s'} P(s, a, s')(R(s, a, s') + \gamma V^*(s')) \end{aligned} \tag{2.3}$$

We get directly the policy according to this value function and it is proved that this policy asymptotic convergence towards π^*

2.3 Experiments

In this part, we tried both VI and PI on all 11 plans of **Grid World** with different parameters, and we discuss some interesting phenomena we met during experiments instead of listing all results. The parameters include values for different boxes and the discount γ we use for dynamic programming. If not mentioned, $\gamma = \mathbf{0.99}$

For the environment we have following components:

Box index with their meaning 0: empty box (black), 1: wal (gray), 2: player (blue), 3: exit (green), 4: object to collect (yellow), 5: death trap (red), 6: non-fatal trap (magenta)

Action index and their meaning:

0: South 1: North 2: West 3: East

We will now discuss some characteristics of these two algorithms and some influences come from environment value we set.

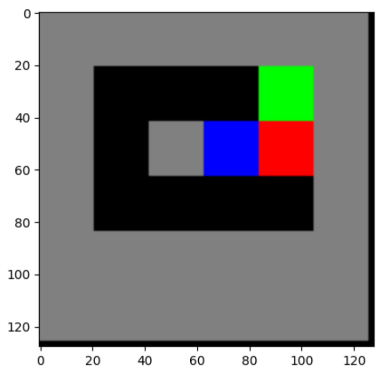
Plan0

This is the easiest plan for grid world model. From figure 2.1 we analyse the behavior of agent (blue box) under different values of empty box . We observe that, PI and VI have exactly the same behavior. When empty box has a small value, e.g., -0.001, our agent try to get as far as possible from the death trap (red box) to finally arrive at the exit (green box). While as we increase the penalty of the empty box, our agent will try to get to the exit as quick as possible as showed in the figure on the left. And we have the following log for training:

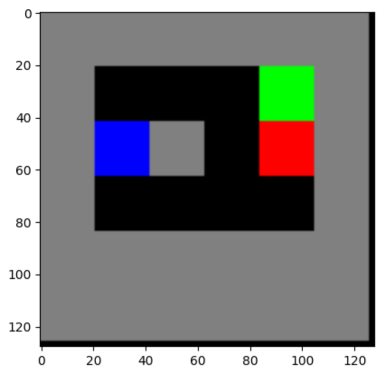
Train Log Plan0

```
env.setPlan("gridworldPlans/plan0.txt",
            {0: −0.1, 3: 1, 4: 1, 5: −2, 6: −1})
Policy converged!
Total iteration: 2141
rsum=0.7, 4 actions

value function converged!
iteration: 25
rsum=0.7, 4 actions
```



(a) PI with 0: -0.1, 3: 1, 4: 1, 5: -2, 6: -1



(b) PI with 0: -0.001, 3: 1, 4: 1, 5: -2, 6: -1

Figure 2.1: Policy iteration on plan 0 with different empty box value

We can conclude that PI take far more iterations than VI to update value function.

Plan3

Figure 2.2 show a plan with only two exits. Here is a train log:

Train Log Plan3

```
env.setPlan("gridworldPlans/plan4.txt",
            {0: −0.001, 3: 2, 4: 1, 5: −1, 6: −1})
Policy converged!
Total iteration: 8041
rsum=0.9750000000000025, 27 actions

value function converged!
iteration: 69
rsum=0.969000000000003, 33 actions
```

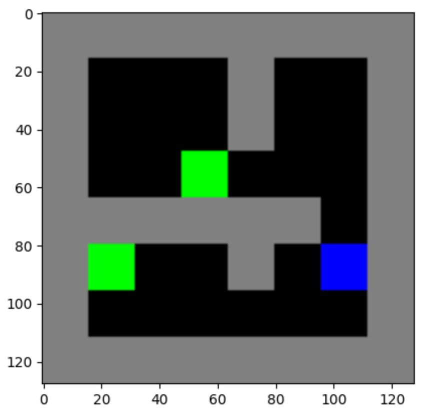


Figure 2.2: Plan3: Only two exit box

As before, PI take more time to converge and there are two interesting points:

- Our agent start from this place will always arrive at the exit above. As the exit above is closer. When we set empty box to zero, two algorithms will also led box to exit above. It is due to the discount γ, as the reward will discount with distance, so we set γ to 1, and then the agent will go to the exit below some times.
- As we set γ to zero, PI may fail to converge while VI can alway converge quickly.

Plan4
Figure 2.3 is an interesting plan. Here is a train log:

Train Log Plan4

```
env.setPlan("gridworldPlans/plan4.txt",
             {0: -0.001, 3: 2, 4: 1, 5: -1, 6: -1})
Policy converged!
Total iteration: 8041
rsum=0.9750000000000025, 27 actions

value function converged!
iteration: 69
rsum=0.969000000000003, 33 actions
```

There are three points to discuss here:

- If the penalty for empty box is high, let's say -1, the model can easily converge to the case that agent collect purple box and exit at green box.
- With a low penalty on empty box and a low γ, the behavior of our agent depends quite a lot on the value for purple penalty box and exit. As the object is between exit and agent and the exit is far away. So If green box value is not enough higher than the purple one, our agent may be blocked in the upper left zone.
- It the discount is sufficiently small, and green provide higher reward than the penalty given by purple there will be no problem that our agent completes the mission.

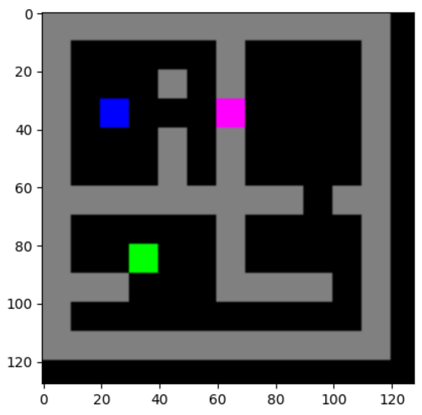


Figure 2.3: Plan4: One object to collect in middle and an exit far away

Plan9

This is the only map that we fail to tackle. As show in Figure 2.4, this map is relatively

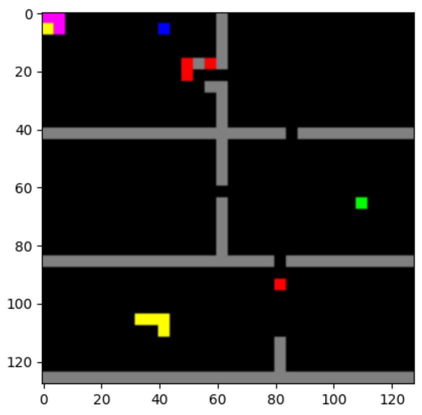


Figure 2.4: Plan9: Big plan with many elements

big with many elements in it, this lead to a huge state space and transition space, that our computer fail to get whole transition space. But we will try to tackle this problem with methods we are going to analyse in the following sections.

Conclusion

For other plans, We concentrate on playing with values, but on some more complicated map. While, the principle is the same. There are some thing to note is that if the penalty on empty box it too high, the agent may try to go directly to red box to finish the round. Sometimes the agent can try to go through the purple one to get a higher gain.

In conclusion, the value of boxes and γ together defined the distribution of our value function and γ talk the agent to look far or near that define the range of a value of one box may influence the others.

Chapter 3

TME3: Q-Learning

3.1 Introduction

In the previous chapter, we play in the case that we possess the MDP model. What shall we do if we do not know the MDP or the MDP is too complicated? We introduce here a very classic method named Q-learning. This is a **value-based**, **model-free** method based on **Q value** defined as below:

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}[\sum_{j=0}^{T}\gamma^{j} r_{t+j+1}|S_t = s, A_t = a] \\ Q^{\pi}(s,a) &= \sum_{s'} P(s_{t+1} = s'|s,a)(R(s,a,s') + \gamma V^{\pi}(s')) \end{aligned} \tag{3.1}$$

Which define the quality to take action a at state s. But as we do not have the MDP for our problem, we will evaluate Q-value instead of value function. We update Q-value by

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_t + \gamma \max_{a' \in A(s_{t+1})} Q(s_{t+1},a') - Q(s_t,a_t)) \tag{3.2}$$

We will test Q-learning and its variants under the **Grid World** environment.

3.2 Algorithms

3.2.1 Q-learning

This is the algorithm we discussed in introduction, we will get samples and update Q by equation 3.2, This is an offline policy, which means that we can use samples generated from other algorithms as long as a sample has tuple (s, a, s', r) so it is more sample efficient. But it has less exploration during training, as we always choose the action that induces maximum Q value.

3.2.2 SARSA

This is an online version of Q-learning, We take action depending on Q value, for example by greedy algorithm. During update, the next state action value depends on the current Q too. Ab=nd this is the action we are going to take for the next step.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)) \tag{3.3}$$

The a_{t+1} is taken according to current Q^π, this method is relatively stable than Q-learning, But as it is on policy, can not use existing samples and it is less sample efficient.

3.2.3 Dyna-Q

Dyna-Q is a hybrid between model-based and value-based method. For our agent, we create a MPD and this MPD is updated together with Q-value. Meanwhile, for each iteration we will also use this MDP to update our Q-value by randomly sample (s, a) from it. Besides equation 3.2, we have

$$\begin{aligned} \hat{R}(s_t, a_t, s_{t+1}) &\leftarrow \hat{R}(s_t, a_t, s_{t+1}) + \alpha_R(r_t - \hat{R}(s_t, a_t, s_{t+1})) \\ \hat{P}(s_{t+1}|a_t, s_t) &\leftarrow \hat{P}(s_{t+1}|a_t, s_t) + \alpha_P(1 - \hat{P}(s_{t+1}|a_t, s_t)) \\ \hat{P}(s'|a_t, s_t) &\leftarrow \hat{P}(s'|a_t, s_t) + \alpha_P(0 - \hat{P}(s'|a_t, s_t)) \quad \forall s' \neq s_{t+1} \end{aligned} \tag{3.4}$$

For each iteration, we will sample K couples of (s, a) and update the value for Q(s,a) like we do in value iteration for one time.

This method should have better propagation of values to avoid model converge in a local optimal.

3.2.4 Q-lambda-learning and eligibility traces

Q(λ) update value by a weighted value from TD(0) to TD(n) where n tends to infinity. We have

$$V_\pi(s_t) = V_\pi(s_t) + \alpha(G_t^{(n)} - V_\pi(s_t)) \tag{3.5}$$

where

$$\begin{aligned} n = 1 \rightarrow G_t^{(1)} &= r_t + \gamma V_\pi(s_{t+1}) \\ n = 2 \rightarrow G_t^{(2)} &= r_t + \gamma r_{t+1} + \gamma^2 V_\pi(s_{t+2}) \\ &\vdots \\ \forall n \rightarrow G_t^{(n)} &= r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V_\pi(s_{t+n}) \end{aligned} \tag{3.6}$$

And we have G(λ)

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{3.7}$$

We wight $G_t^{(n)}$ by a decreasing weigh 1, λ, λ^2,..., λ^n and the sum of them is $\frac{1}{1-\lambda}$ To calculate this update, we introduce eligibility traces. All $G_t^{(n)}$ we calculate now is a froward view and we need to wait until the end of an episode to do the the propagation of values. With eligibility traces, we transfer the update to a Backward view that we update for all value we have met for each step we take. At the end of episode, these two methods are equivalent. With eligibility traces lead to a quicker convergence than TD(0).

3.3 Experiments

Plan1

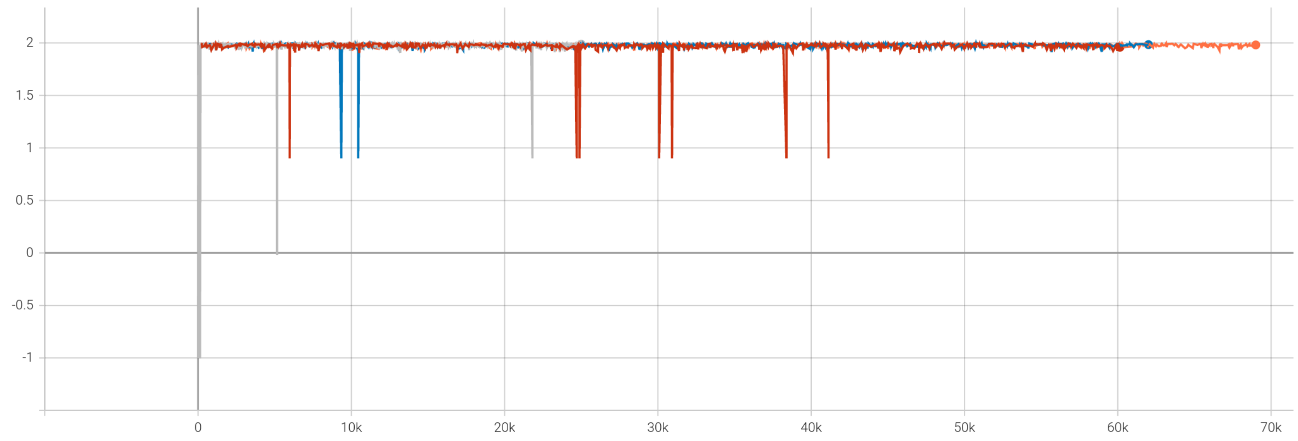


Figure 3.1: Grid World Plan1, **blue**: rsum of **Q-learning**, **orange**: rsum of **SARSA**, **red**: rsum of **Dyna-Q**, **gray**: rsum of **Q(λ)** with eligibility traces. Horizontal axis: number of episode.

At the first time, we implemented Q-learning and SARSA, Dyna-Q, $Q(\lambda) - learning$ and run it on the plan1 of Grid World. We can have default setting for Grid World as rewards: 0: -0.001, 3: 1, 4: 1, 5: -1, 6: -1. And all of these for methods converge quickly to the optimal point as this plan is relatively easy.

Plan5

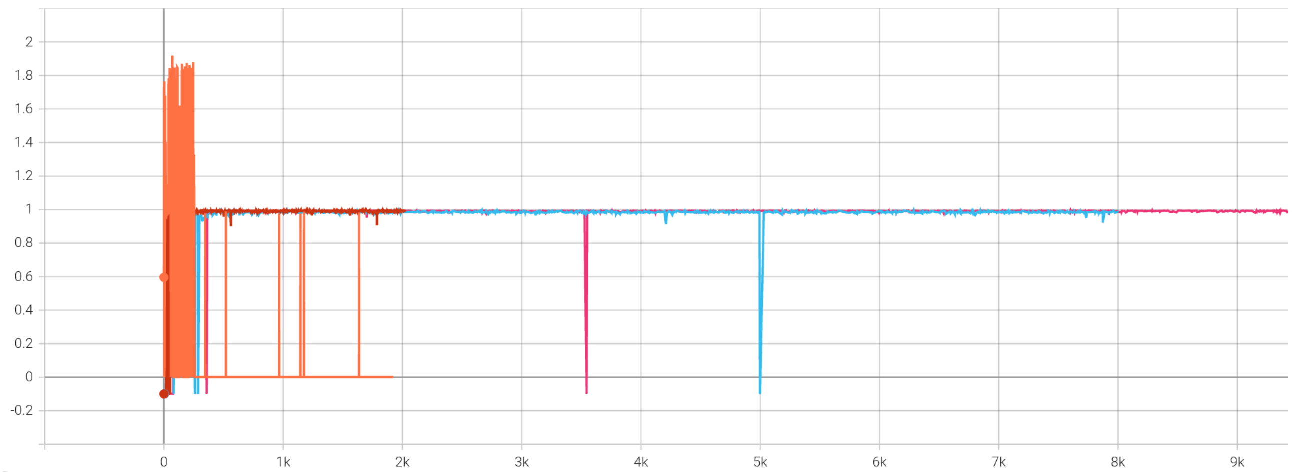


Figure 3.2: Grid World Plan5, **blue**: rsum of **Q-learning**, **orange**: rsum of **Dyna-Q**, **pink**: rsum of **SARSA**, **red**: rsum of **Q(λ)** with eligibility traces. Horizontal axis: number of episode.

We find out that all of these methods except Dyna-Q converge to very quickly to local optimal as they do not go to lower part of the map to fetch the object but go directly to the exit. While, Dyna-Q achieve some good performance at the beginning but it degrades 400 episodes and we do not know the reason for this phenomena.

Plan9

We didn't finds a way to deal with plan 9 in the previous chapter, we try these four methods on plan 9 to evaluate their performance. Finally, we find some methods to tackle this problem, but as we show in the result of plan5, only Dyna-Q can achieve the global optimal sometimes

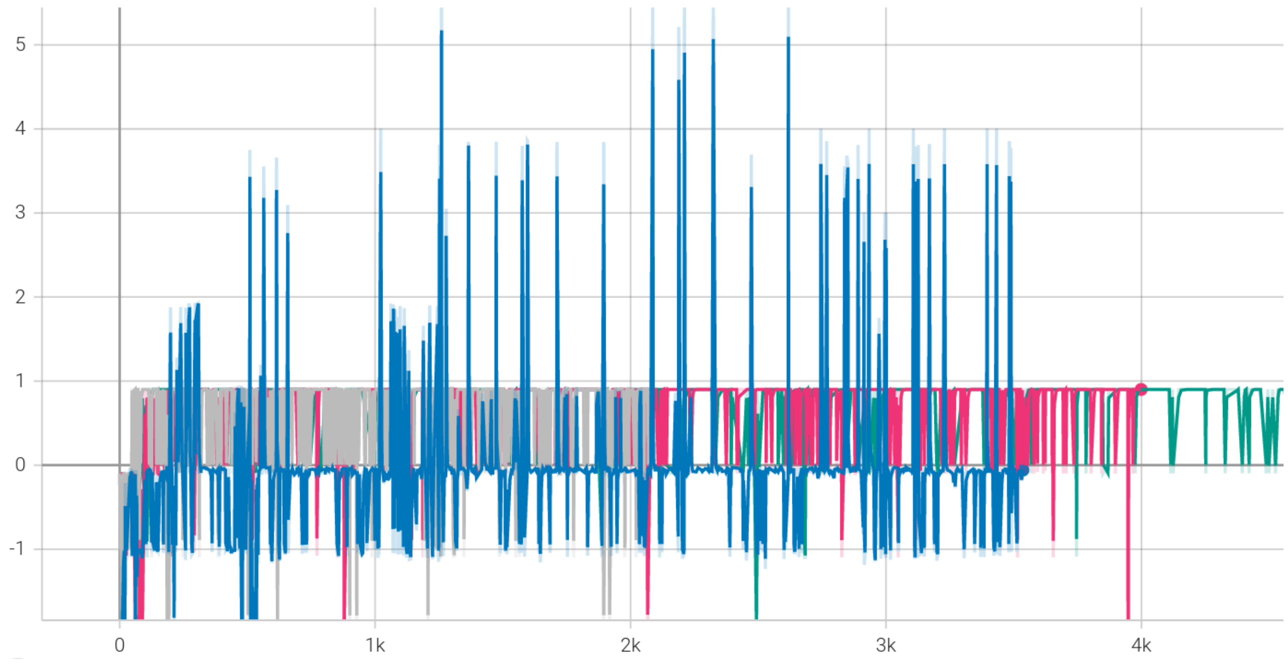


Figure 3.3: Grid World Plan9, **pink**: rsum of **Q-learning**, **blue**: rsum of **Dyna-Q**, **green**: rsum of **SARSA**, **gray**: rsum of **Q(λ)** with eligibility traces. Horizontal axis: number of episode.

and other methods tends to converge to a local maximal. And here is an example of Dyna-Q.

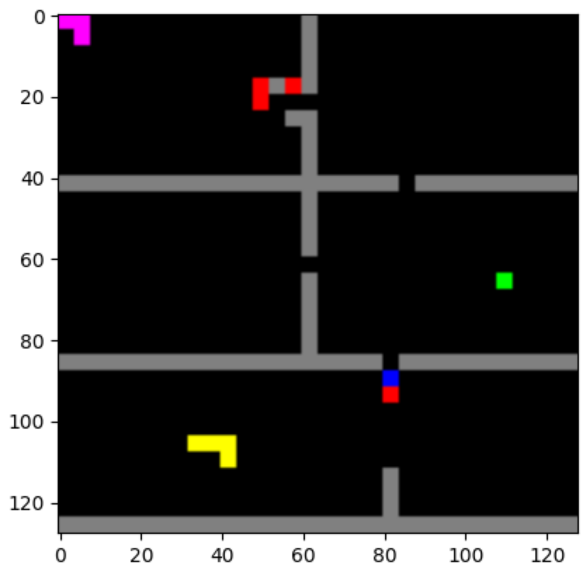


Figure 3.4: Example of Dyna-Q go to lower part of the map to get the objects and then go back to exit on the upper part of the plan.

Conclusion

We implemented and analysed four value based method on Grid Word plans. And for now, all of method based on table of values. There are some problems for this kind of method as we show during the experiments.

- Exploration difficulty: Model tands to converge to sub-optimal.
- Large dimension difficulty: If the dimension of problem is very large, we will have problem to storage all state and action and the value we learned has limited generalisation.

In the next chapter, We are going to explore some approximation of value based method by using neural network. Those methods have achieved very surprising achievements in recent years.

Chapter 4

TME4: Approximate Q-Learning

4.1 Introduction

One cannot always apply the algorithm of the previous chapter because the environments can be too complicated to create a table. Hence, one seeks to an approximation of the tabular Q-learning. This is the motivation of deep Q-Learning. Here, we approximate the Q-function by an neural network and train this network and we represent state by a vector generate by a function Φ which we create to transfer a state to features of a state. For example in Grid Word, The position of different boxes.

4.2 Algorithms

4.2.1 DQN: Deep Q Learning

Similarly to the previous chapter, one defines the target value y by the Bellman equation:

$$y = r + \gamma \cdot (1 - \text{done}) \cdot \max_{a'} \hat{Q}(s', a') \tag{4.1}$$

Target Network

In equation 4.1, $\hat{Q}$ is a target network which we copy value from Q value network each K iteration to stable the evaluation of parameters and reduce the correlation caused by samples. To obtain the action from the Q-values, we apply again the $\epsilon-$greedy strategy.

Mini Batch Optimisation

The optimization of the Q-network is based on the data of a sampled mini-batch from the memory. In the memory, all transitions (consisting of state, action, reward, next state, done) are stored. Hence, the DQN is an off-policy learning algorithm. Then one optimizes the Q-network on this mini-batch. This can help to accelerate the optimisation process and to reduce correlation between samples. Samples will also be used several times to avoid 'forgetting'.

4.2.2 PER: Prioritized Experience Replay

A priori, the transitions are uniformly sampled uniformly from the memory. This is called experience replay. However, we can also prioritize those transitions whose $TD(0)-$error is larger, while sampling the mini-batch. In this way, one can concentrate on the important transitions during the learning step. Than we use a trick on weight to reduce the bias caused by the change of distribution which samples follow . This version is called prioritized experienced replay. As we attribute each sample a weight, usually we use **cumulative sum** and an uniform distribution to sample. But this can lead to a complexity of O(N), where N is the number of samples in our buffer. For Deep-Q learning, sample number can be very huge, so we turn to **Sum Tree** structure that is a an efficient algorithm with a complexity of O(log(N)). There is a very good blog [2] about it.

4.2.3 Double DQN

As we know that

$$\mathbb{E}[max(X_1, X_2)] \geq max(\mathbb{E}[X_1], \mathbb{E}[X_2])$$

DQN's temporal differential update process over-estimated Q value. As the equation below

$$\mathbb{E}[r_t + \gamma \max_{a \in \mathbb{A}(s_{t+1})} Q_{\theta^-}(\phi(s_{t+1}), a)] \geq \mathbb{E}[r_t] + \gamma \max_{a \in \mathbb{A}(s_{t+1})} (\mathbb{E}[Q_{\theta^-}(\phi(s_{t+1}), a)])$$

So we change a little bit the update process to the right part of equation

$$\delta_t = r_t + \gamma Q_{\theta^-}(\phi(s_{t+1}), arg \max_{a \in \mathbb{A}(s_{t+1})} Q_\theta(\phi(s_{t+1}), a)) - Q_\theta(\phi(s_t, a)) \tag{4.2}$$

4.3 Experiments

We will apply these two versions on the "CartPole-v1" and on the "LunarLander-v2" environments.
For the Cartpole environment, we used the following hyper-parameters:

- We used neural network with one hidden layer of dimension 200 as our Q-network. The tanh-function was used as activation. The network was optimized by stochastic gradient descent with a learning rate of $5 \cdot 10^{-3}$.
- We set the discount rate $\gamma = 0.9$.
- To increase exploration, we used the ϵ-greedy strategy. We set $\epsilon = 0.15$ used a decay rate of 0.99999.
- We optimized the network every 20 events. For this, we used a batch size of 100. Furthermore, we updated the weights of the target network every 100 events. The maximum length of an episode is limited to 300.

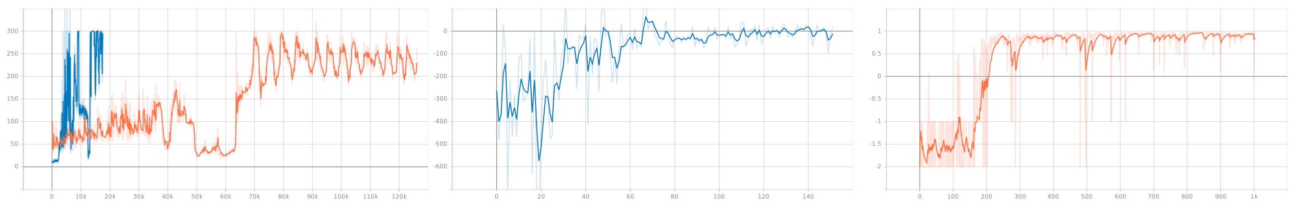
For the LunarLander environment, we changed some parameters:

- The neural network consists of two hidden layers of dimension 32. Furthermore, the learning rate was reduced to $3 \cdot 10{-}4$.
- The learning step is performed after every action and batch size is reduced to 64. Additionally, the maximum train length is 500.

We ran only the prioritized version on the LunarLander environment because this algorithm is more powerful which is crucial on the more complicated environment.
We also applied the DQN on the plan 4 of Gridworld as in fig. 2.3. The rewards of the states are the same as in section 2.3. For the learning, we used the following hyper-parameters:

- The Q-network has one hidden layer of dimension 200 and the tanh as an activation. The learning rate is set to $1 \cdot 10^{-3}$.
- We use a discount of 0.99.
- Again, an ϵ-greedy strategy is employed with $\epsilon = 0.2$ and and a decay of 0.99999.
- We optimize every 10 events with a batch size of 64. The weights of the target network are updated every 100 events. The maximal train length is 1000.
- We use the experience replay version.

By using the parameters from above, we obtained the reward curves as can be seen in fig. 4.1. One can see that the prioritized version of DQN clearly outperforms on the CartPole environment. The prioritized DQN performs also well on the Lunar Lander environment. However, we will later see some more powerful algorithms which can achieve significantly higher rewards. At last, the Gridworld problem is quite easy for the DQN and is solved very fast by the algorithm.



(a) CartPole-v1 (b) LunarLander-v2 (c) Gridworld, plan 4

Figure 4.1: Reward curve in the respective environments
orange: experienced replay, **blue**: prioritized experienced replay

Comparison between DQN and Double DQN

We compare these two algorithms in Lunar Lander mission. Because their performances are similar in Grid world and Cart pole and they both have PER. We notice that Double DQN

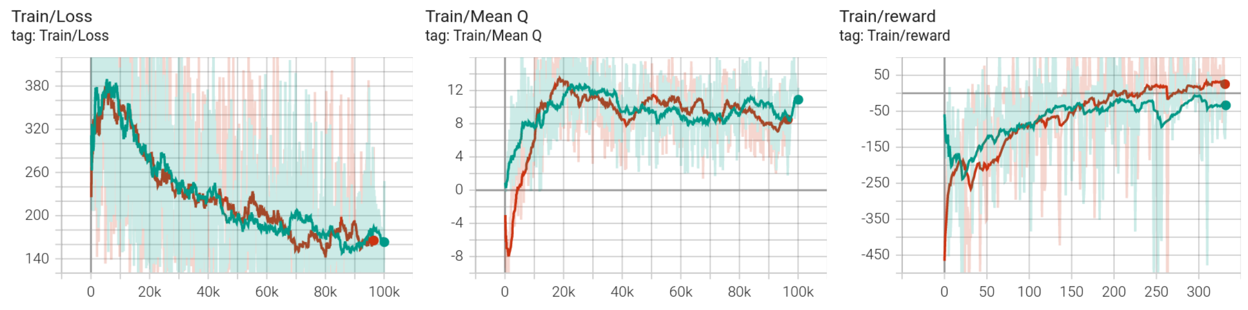


Figure 4.2: DQN + vs Double DQN in Lunar Lander. **Dark blue**: lr 0.001. **red**: lr 0.0001. **blue**: lr 0.005

has better performance and it continues to improve. The mean Q value it gives is relatively lower than DQN. It is able to land the aircraft to land smoothly on the ground most of the time.

Discuss

Even DQN is a very cool algorithm with better generalisation and can be used for complicated task, it does not mean it can do better for all missions. For example on Gird World, it

(a) Go Down example (b) Land example

Figure 4.3: Lunar lander with Double DQN + PER example

converge quickly to sub-optimal as table version. And as all value based methods, it could lead to very strong change of strategy.

Grid World plan5, learning rate tuning Even with PER, the Mean value of Q func-

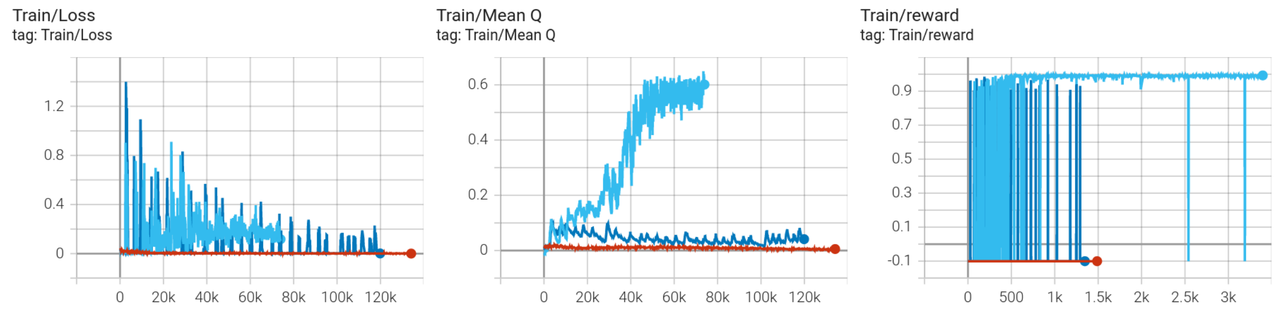


Figure 4.4: DQN + PER on plan5 of Grid World. **Dark blue**: lr 0.001. **red**: lr 0.0001. **blue**: lr 0.005

tion depends quite a lot on the learning rate of the model and this lead to a huge difference on agent's behavior. With big learning rate of blue, the reward change brutally for each episode and with little learning rate in red, it is stacked to a local region and agent does nothing. while for the blue one, it go into a local optimal. So in the following chapter, We are going to introduce the policy based methods and hybrid of policy based and value based methods which has smoother policy optimisation process.

Chapter 5

TME5: Policy Gradients

5.1 Introduction

In the previous chapter, we trained the algorithm to learn the Q-values. The policy was then deduced by those Q-values, e.g. through an $\epsilon-$greedy strategy. Now, one wants the algorithm to learn directly the policy. This will be the challenge of the actor. The actor takes a state as the input and computes a probability for each action. Then, one can sample an action from this distribution. In this way, we obtain stochastic policy. Additionally, there is the critic network which learns the value function and criticizes the actions made by the actor. In contrast to the DQN, the actor-critic algorithm is on-policy and hence the optimization is based only on the transitions sampled from the current policy.
There are several possibilities to implement the target value of the critic network and the advantage, which is used for optimising the actor. We have implemented three version of the actor-critic algorithm:

- $TD(0)$-learning
- Roll-out
- Roll-out with generalized advantage estimation

For the $TD(0)$-learning, we defined the target as in the chapter before according to the Bellman equation. The advantage is approximated as following:

$$A(s,a) = Q(s,a) - V(s) \approx r(s,a) + \gamma \cdot (1 - \text{done}) \cdot V(s') - V(s) =: \hat{A}(s,a)$$

We used a target network to compute the values.
For the roll-out versions, we consider a whole trajectory and compute the exact discounted returns which are then used as the target values. The advantage is defined as the difference of these discounted returns and the current state values which are calculated by a target network. The generalized advantage estimation discounts additionally the advantage itself by an parameter λ and is computed iteratively as following:

$$\hat{A}(s_t, a_t) = \gamma \cdot \lambda \cdot (1 - \text{done}) \cdot \hat{A}(s_{t+1}, a_{t+1}) + \text{TD(0)-error}$$

In all cases, the actor network is then optimized via gradient ascent of the following loss:

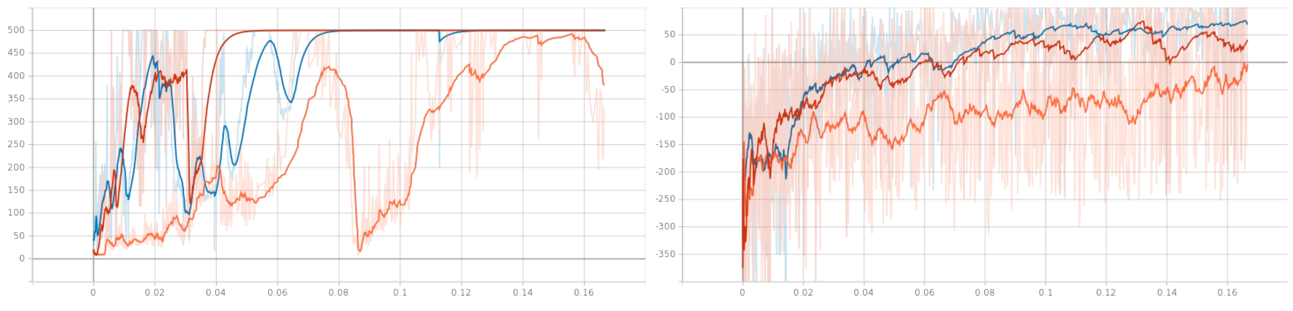
$$J(\theta) = \sum_{(s,a)\in\tau} \log \pi(a|s) \cdot \hat{A}(s,a)$$

5.2 Experiments

We used the following hyper-parameters for the implementation of these three versions :

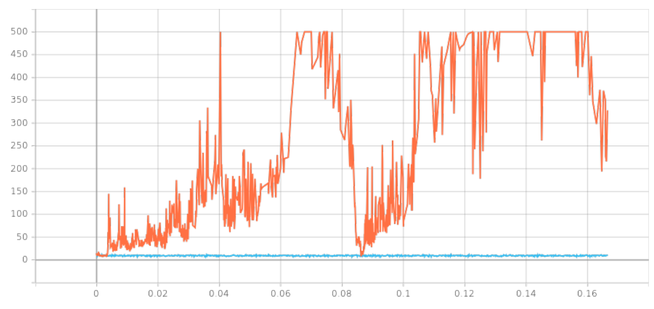
- The actor and the critic network are 3-layer neural networks with hidden dimensions 128 and 256. We used the tanh function as an activation. For the critic network, the Huber loss was used. We optimized the model with the Adam optimizer and a learning rate of $1 \cdot 10^{-3}$.
- We did an soft-update for the weights of the target network with the Polyak parameter $\rho = 0.99$.
- $\gamma = 0.99$, $\lambda = 0.99$
- We added the entropy loss to the actor loss with a coefficient of $1 \cdot 10^{-4}$.
- We trained the model after each episode and we ran the algorithm for ten minutes for each version.
  It makes more sense to compare the performance relative to the time, since for example $TD(0)$-algorithm does just ten actions in each of the first 400 episodes. That means that the algorithm does lots of actions in a very short period of time. On the other hand, the roll-out version reaches 500 actions, the maximum, the first time after 184 episodes. Hence, it does less episodes in a longer period of time.

We tested these algorithms on the "Cartpole-v1" and on the "LunarLander-v2" environment and visualized the results in fig. 5.1. We see that the roll-out versions outperform clearly the $TD(0)$-learning algorithm. Furthermore, one can deduce that the impact of the generalized advantage is not too large since the two roll-out algorithms have comparable rewards in both environments.
We ran also the $TD(0)$-Learning algorithm without the entropy loss. In this case, it can happen that the algorithm does not work at all, as can be seen in fig. 5.2. As the KL-divergence is just constant in this case, one can deduce that the algorithm does not change the policy. With the addition of the entropy term, one forces the algorithm to explore more policies. We see that this exploration is crucial for the $TD(0)$-Learning.

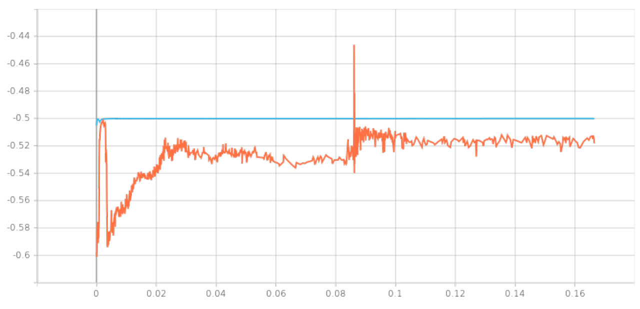


(a) CartPole-v1 (b) LunarLander-v2

Figure 5.1: Reward curve over 10 minutes.
orange: $TD(0)$-Learning, blue: Roll-out with usual advantage, red: Roll-out with generalized advantage



(a) Reward curve



(b) Kullback-Leibler divergence

Figure 5.2: orange: $TD(0)$-learning with entropy loss, blue:$TD(0)$-learning without entropy loss

Chapter 6

TME 6: Advanced Policy Gradients

6.1 Introduction

The actor-critic algorithm has unfortunately several disadvantages. The policy is very sensitive, i.e. little changes in the parameters can significantly change the resulting probability distribution. Furthermore, the algorithm suffers from a large variance and low efficiency. These problems are tackled by the so-called Trust-Region-Policy-Optimization (TRPO) algorithm. However, it uses second-order derviatives and is therefore very slow.
The Proximal-Policy-Optimization (PPO) algorithm is motivated by the TRPO and shows a comparable performance to that algorithm, while using only fist-order derivatives.
The general idea of the PPO is that one wants to update the weights of the policy in following way:

$$\theta_{k+1} = \arg\max_{\theta} \frac{\pi_\theta\left(a|s\right)}{\pi_{\theta_k}\left(a|s\right)} \cdot A^{\pi_{\theta_k}}(a,s)$$

In addition, one can modify this update rule by adding the Kullback-Leibler (KL) constraint or by using a clipped version.
The advantage is computed as in version generalized advantage estimation from the previous chapter 5.
We have implemented the following three versions of the PPO: the normal version, the clipped version and the version with an adaptive KL-penalty.

6.2 Experiments

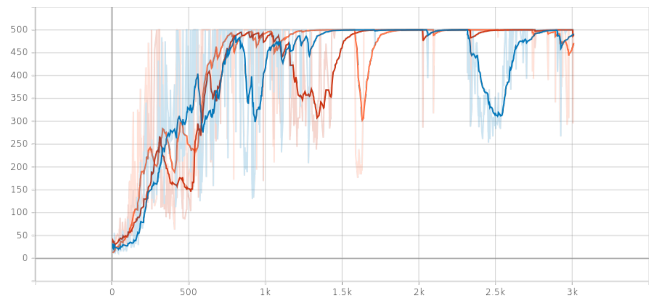
The algorithms were tested on the "CartPole-v1" and on the "LunarLander-v2" environment where we used the following hyper-parameters:

- For the actor and the critic network, we used a neural network consisting of two hidden layers of dimension 64 with an tanh activation which was optimised by the Adam optimizer with a learning rate of $2 \cdot 10^{-4}$. The Huber Loss is used as loss for the critic network in order to prevent exploding gradients.

- We used the following values as the discount rate and as the parameter for the advantage estimation :
$\gamma = 0.999$, $\lambda = 0.99$

- As suggested in the paper which introduces the PPO, we used the following PPO-related hyper-parameters:
  $\beta = 1$, $\delta = 0.01$, $\epsilon_{clip} = 0.2$
- The model was trained for five epochs every 500 steps. A testing of 5 episodes was employed every 500 steps. The maximal length of an episode is 500 steps.

One can see the results of those experiments in fig. 6.1.As can be seen, all three versions perform well and have comparable rewards.
One can also add an entropy loss to the actor loss in order to increase the exploration of the agent. This was employed with an entropy coefficient of $1 \cdot 10^{-4}$ on the "LunarLander-v2" environment. As can be seen in fig. 6.2a, there is no impact of the entropy loss. This is a significant difference to the previous Actor-Critic algorithms (see fig. 5.2).
We compared also the performance of the PPO with the KL-penalty with Actor-Critic model with the generalized advantage estimation (as in the previous chapter 5). We ran both algorithms on the "LunarLander-v2" environment for ten minutes. The rewards are plotted in fig. 6.2b. One sees that the PPO algorithm clearly outperforms the Actor-Critic model.


(a) CartPole-v1

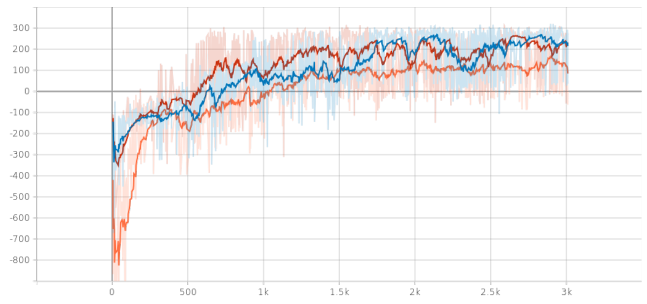
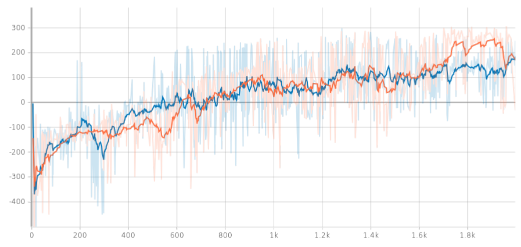

(b) LunarLander-v2

Figure 6.1: Reward curve over 3000 episodes.
orange: PPO with KL-penalty, blue: clipped PPO, red: normal PPO


(a) blue: Clipped PPO with entropy loss,
orange: Clipped PPO without entropy loss

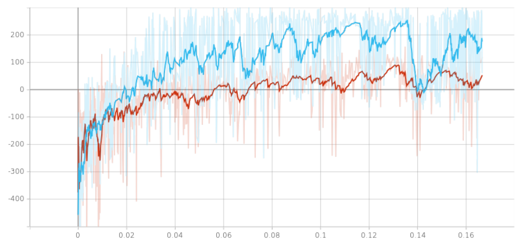

(b) blue: PPO with KL-penalty,
red: Actor-Critic with generalized advantage

Figure 6.2

Chapter 7

TME 7. Continuous Actions

7.1 Introduction

In this chapter, we consider environments where the action space is continuous. This is a fundamental difference to the previous chapter because we cannot apply the same algorithms anymore. However, one can modify the algorithms for the discrete action spaces in order to apply them for the continuous cases.
First, we will consider the Deep Deterministic Policy Gradient (DDPG) algorithm. This algorithm also uses an actor and a critic network. However, the output of the actor network does not represent the probabilities of the actions (as in the discrete case). It represents directly the continuous actions. So by construction, the policy is deterministic which will limit the performance of the algorithm. The critic network learns the Q-values of state-action-pairs. The actor is then optimized by maximizing

$$\frac{1}{|B|} \cdot \sum_{s \in B} Q_\phi\left(s, \pi_\theta(s)\right)$$

This algorithm is hence quite intuitive and simple. Note that the learning step is based on data sampled from a memory. So, the DDPG is off-policy.

7.2 Experiments

We test the algorithm on three environments with an continuous action space and use the following hyper-parameters:

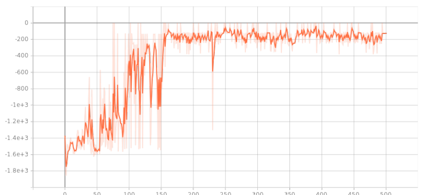
- The actor and the critic network are 3-layer neural networks with an hidden dimension of 256 and the ReLU as an activation. As the final activation of actor network, we use the tanh-function scaled by the maximum of the action space. This makes sense since the action space is in these three environments symmetric around 0.
  We optimize both networks with the Adam optimizer and a learning rate of $1 \cdot 10^{-3}$. For the defining the target in the algorithm, we use a target network for the actor and the critic whose weights are updated via Polyak with an parameter $\rho = 0.995$.

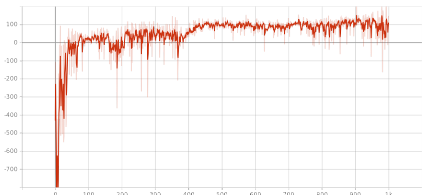
- We use a discount rate of 0.99.

- To increase exploration, we add an sample of the Ornstein-Uhlenbeck-process to the action.We set the parameters of the stochastic process as following:
  $\mu = 0$, $\theta = 0.15$, $\sigma = 0.2$.

- We use a batch size of 100 and optimize 50 times every 200 events. The maximum train length is 200 actions and we run the algorithm for 1000 episodes.

The results can be seen in fig. 7.1. The performance on the Pendulum environment is satisfying. However, we see that the algorithm on the Lunar Lander environment cannot perform better than a reward of 100. So, the performance is limited by this algorithm. This is a motivation for the algorithm of the next chapter chapter 8 where we use an algorithm which can achieve higher rewards (see fig. 8.2a).
We also have to note that the performance on the Mountain Car environment could not always be achieved with these hyper-parameters. That is why we tuned them a bit. We found out that an increase of the standard deviation of the Ornstein-Uhlenbeck process can improve the performance as can be seen in fig. 7.2a. However, this cannot be generalized to other environments. For example, we see that the performance on the LunarLander environment decreases slightly and varies more when using this standard deviation (see fig. 7.2b).


(a) Pendulum-v0


(b) LunarLanderContinuous-v2

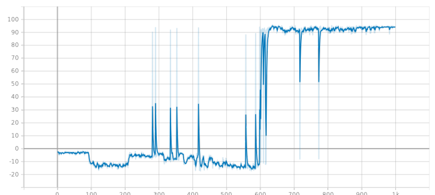
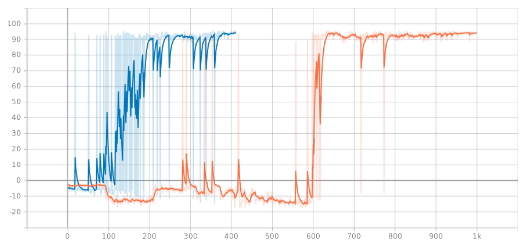

(c) MountainCarContinuous-v0

Figure 7.1: Reward over 1000 episodes of the DDPG in the respective environments


(a) MountainCarContinuous-v0

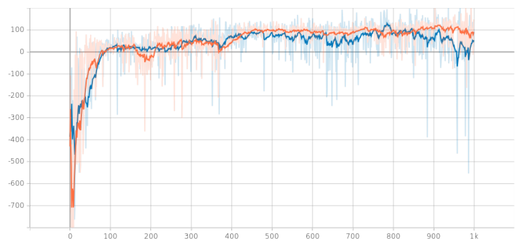

(b) LunarLanderContinuous-v2

Figure 7.2: Reward over 1000 episodes of the DDPG in the respective environments
orange: $\sigma = 0.2$, blue: $\sigma = 0.3$

Chapter 8

TME 8. Soft Actor-Critic (SAC)

8.1 Introduction

In this chapter, we consider another algorithm which can handle continuous action spaces. It is called the Soft Actor-Critic (SAC) algorithm.
A disadvantage of the DDPG-algorithm from the previous chapter 7 is that the policy is deterministic. Since we cannot learn the probabilities of each action (as in the case of the discrete actor-critic algorithm), we must assume that the actions follow a given probability distribution. For this, we take the Gaussian distribution $\mathcal{N}\left(\mu_\theta(s_t),\, \sigma_\theta(s_t)\cdot I\right)$ and we let the algorithm learn its parameters $\mu_\theta(s_t)$ and $\sigma_\theta(s_t)$. These parameters will be the output of the actor network (more precisely the output will be $\mu_\theta(s_t)$ and $\log(\sigma_\theta(s_t))$).In addition, we have two critic networks which learn the Q-values of state-action-pairs (as in Double Q-Learning). Additionally, we use an entropy term for the target value of the critic networks with an entropy coefficient α. So, the target value y of the critic networks is defined as:

$$y = r + \gamma \cdot (1 - \text{done}) \cdot \Big(\min_i Q_i(s', \widetilde{a}) - \alpha \cdot \log \pi_\theta(s', \widetilde{a})\Big) \qquad , \widetilde{a} \sim \pi_\theta(s')$$

And then actor network is trained with the following loss:

$$\frac{1}{|B|} \cdot \sum_{s \in B} \Big(\min_i Q_i(s, \widetilde{a_\theta}) - \alpha \cdot \log \pi_\theta(s, \widetilde{a_\theta})\Big) \qquad , \widetilde{a_\theta} \sim \pi_\theta(s)$$

There are two version of the SAC. Either one fixes α or one additionally learns α with the loss:

$$J(\alpha) = -\alpha \cdot \Big(\log \pi_\theta(s', \widetilde{a}) - \mathcal{H}_0\Big) \qquad , \widetilde{a} \sim \pi_\theta(s)$$

where $\mathcal{H}_0$ is a target entropy.

8.2 Experiments

We used the following hyper-parameters for running the algorithm on the different environments:

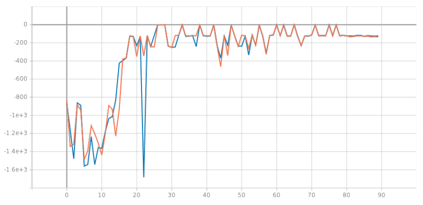
- The policy and the critic networks consist of three layers with hidden dimension 512 and ReLU as activation. As the final activation of the policy network, we used again the scaled tanh-function. To compute the target values, we used a target network of the critic network.
  We optimized the networks with an Adam optimizer using a learning rate of $1 \cdot 10^{-3}$.

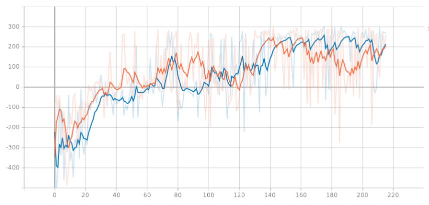
- In case of having an adaptive temperature, we optimize the α-loss with Adam optimizer of the same learning rate too. Then, we set the target entropy $\mathcal{H}_0 = -dim\,(\mathcal{A})$, where $\mathcal{A}$ is the action space of the environment. This was proposed by the original paper of the SAC. Furthermore, we set the initial $\alpha = 1$.
  If we consider α to be fixed, we set $\alpha = 0.2$.

- We use a discount $\gamma = 0.99$ and the Polyak parameter of the Soft-update $\rho = 0.99$.

- We do first 1500 random actions to fill the memory. Afterwards, we perform one gradient step after each action.
  We use a batch size of 128.

Using these parameters, we obtain the following reward curves in fig. 8.1 for the respective environments.
With the hyper-parameters from above, the SAC does not converge in the Mountain Cars environment. But we have seen in fig. 7.2a that the exploration is crucial for this environment. That is why we added a noise term (via an Ornstein-Uhlenbeck process with $\sigma = 0.5$) to the action. We performed a testing step every five steps in order to evaluate the performance without the noise.
In fig. 8.2b, one can also see that the value of α is approximately 0.1 during almost the whole running time.
We compared the DDPG and the SAC on the "LunarLanderContinuous-v2" environment. The reward curve relative to the time can be seen in fig. 8.2a. We can deduce from there that the SAC can reach higher rewards and is hence more powerful than the DDPG on complicated environments.


(a) Pendulum-v0


(b) LunarLanderContinuous-v2

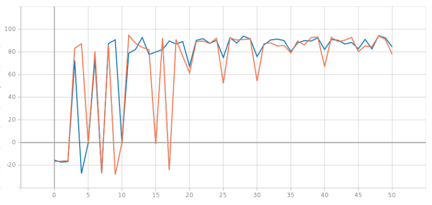
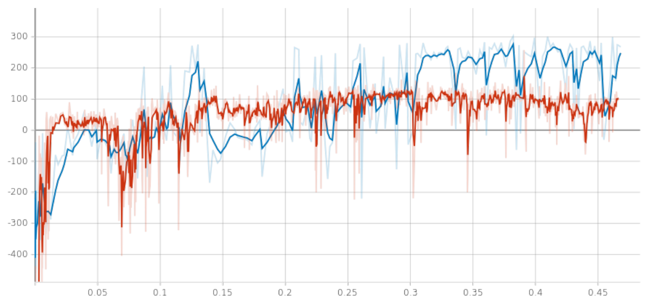
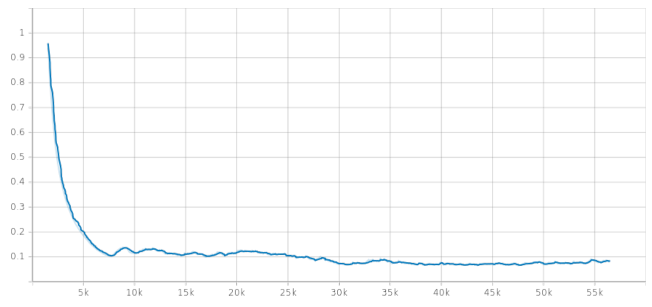

(c) MountainCarContinuous-v0

Figure 8.1: Reward of the SAC in the respective environments.
Blue: Non-adaptive SAC, orange: adaptive SAC


(a) Comparison between non-adaptive SAC (blue) and DDPG (red) on "LunarLanderContinuous-v2"


(b) α w.r.t. number of steps when running the adaptive SAC on "LunarLanderContinuous-v2"

Chapter 9

Conclusion

In this report, we have seen many methods which try to solve several problems of reinforcement learning. These techniques have different approaches (e.g. model-based vs model-free or value-based vs policy gradients). We also discussed two algorithms which can be applied to continuous action spaces. In the end, we have seen some outperformances among those algorithms. However, one must note that the choice of the algorithm depends crucially on the environment and there is no holy grail.

During these sessions of TME, We have not only a better understanding of what we have learnt during course by implementing them by hand, but also have we gotten more familiar with the reinforcement learning community by going through technical blogs, reading source codes on github.

Reinforcement learning is a very dynamic field, what we have learnt in this course is a very good starting point for our future study. Besides getting more experience with tuning parameters and try different algorithms, we should also try to understand the mathematics behind all of them, which will help us to tackle unseen problems or even contribute to the improvement of this field of research.

Finally, we have tested nearly all methods including bonus methods except Noisy-DQN, Q-prop due to lack of time. Thanks to this course we have now possessed a great base of reinforcement learning.

Bibliography

[1] *Cours source, RLD.* https://dac.lip6.fr/master/rld-2021-2022/.

[2] *SumTree introduction in Python.* https://adventuresinmachinelearning.com/sumtree-introduction-python/.