Report for AI3603: Homework II

Chen Jingtao

November 13, 2025

Contents

1 Reinforcement Learning in Cliff-walking Environment

In the Cliff-walking Environment, we use three differnet frameworks to train an agent to guide us through the cliff-map.

1.1 Code Implementation

In *agent.py*, we define a *BaseAgent* as the basic agent class, and then we implement the different *learn()* for three agents respectively.

For Sarsa, we use the formula $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$ to learn from experience, and for each time we learn as we heading our current status towards the next status we choose in *learn()*.

For Qlearning, we use the same formula as Sarsa to learn, but the difference is that out learning does not depend on the next action we choose, and each time we just select the maximum status among all possible next-action status.

For DynaQ, base on Qlearning, we use a brand new method to learn. We store the past status-pair that we have explored, and each time we learn, we sample randomly from the history and refresh it with the formula.

1.2 The Differnce Between Sarsa and Qlearning

We can clearly see the difference by the two graphs below:

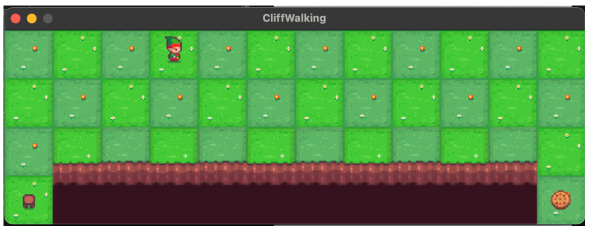

Figure 1: Sarsa Agent

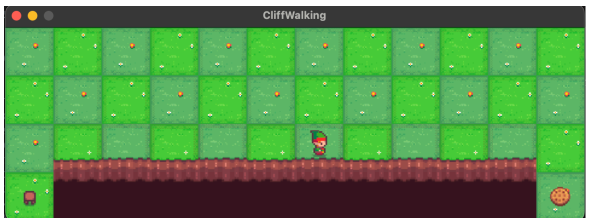

Figure 2: Qlearning Agent

The route found by Sarsa-agent will lead the character to the top of the map and then go right, while Qlearning-agent will lead the character directly right forward to the destintion. So why this distiction occurs?

In fact, Sarsa-agent will try its best to avoid the near-cliff block, since the risk of these blocks are too high. In Sarsa learning part, we need to choose a next-status and then step into it, so the agent will put high value on the risk of the next-status. Therefore, it will choose a path that is most safe, which lead us far away from the cliff instead of considering the shortest path.

In another view, we see Qlearning-agent to be a quite ”greedy” agent. Each time the agent learn, it doesn’t take the next action of the next atutus into consideration, and only focus on the maximum reward it may have. So it will ignore the possible risk of going near cliff, and choose the shortest path instead. That’s why it selects a quite different path than Sarsa.

1.3 The Training Efficiency between model-based RL and model-free algorithms

Let’s first check out the training effiecncy of model-free algorithms:

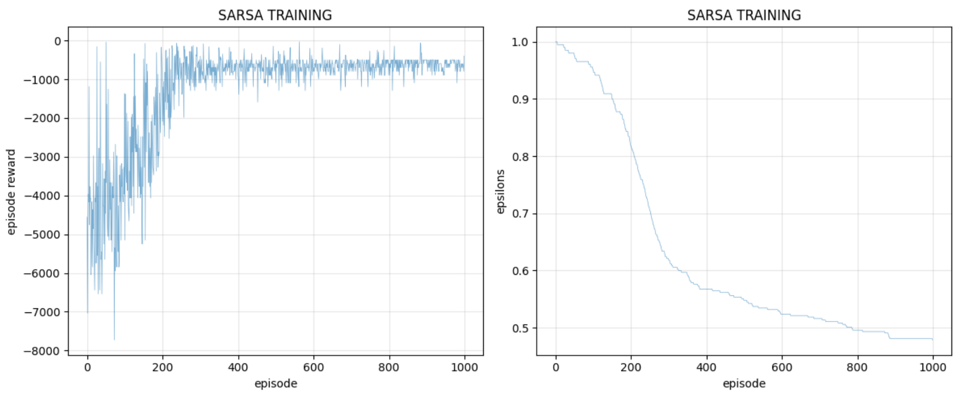

Figure 3: Sarsa Training

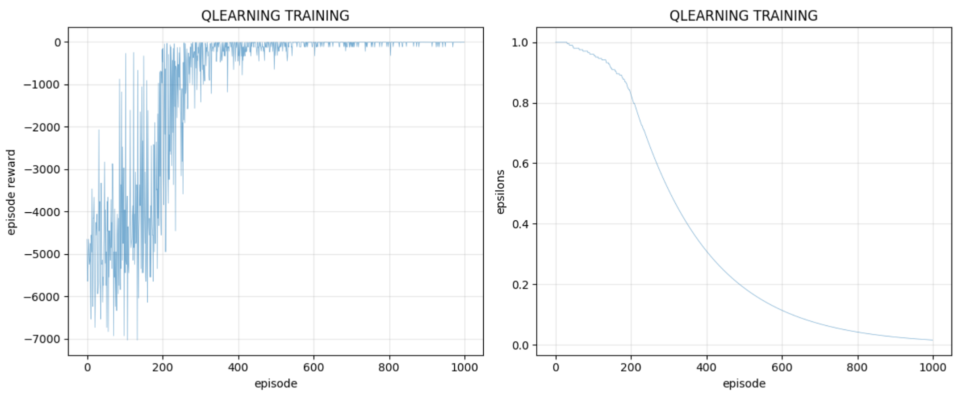

Figure 4: Qlearning Training

From the plots above we can see the episode reward converge to the result at about 200-300 episode. Even if the two frameworks' final reward is a little different as what we have discussed at the last section, their converging trend and is similiar. Besides, it could be noticed that the curve of Sarsa is not stable ,floating after converging.

And as follows are the performance of dynaq:

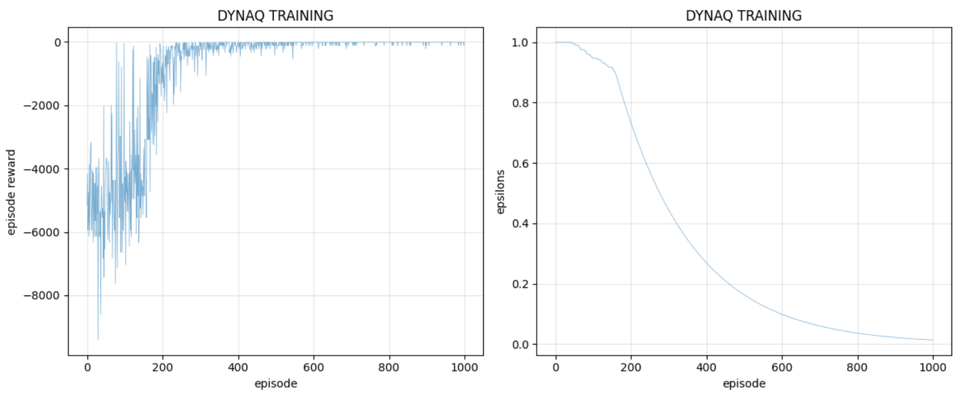

Figure 5: Dynaq Training

From the plots we can see the Dynaq, which is a model-based RL method, converges at about 200 episode, quite faster than the model-free algorithms. It learns very quickly at 150-200 episode, and then it's very stable after converging.

So we can conclude that the model-based RL is more efficent at training than the model-free algorithms, while have a better performance at the quality of the agent we finally get.

1.4 Videos

I attached the gym-render game video in *./videos*, including *sarsa.mp4*, *qlearning.mp4* and *dynaq.mp4*.

2 Deep Reinforcement Learning

In this section we implement, train and tune a Deep Q-Network, to complete the lunar-landing game.

2.1 Code Implementation

Base on the provided code, I adjust some part of the code to make it more efficient.

- I add the mps-device support to train the model on mac-os.
- I import a τ to apply soft update to the target network, which makes the update smoother
- I add the video-recored part
- I once added a reward shaping part, but finally I give it up (Have been commented so far)

2.2 Hyperparameters

My final h-parameter deployment is as follows:

```
python dqn.py \
    --total-timesteps=500000 \
    --learning-rate=0.00025 \
    --buffer-size=80000 \
    --gamma=0.99 \
    --tau=0.005 \
    --start-e=1.0 \
    --end-e=0.01 \
    --exploration-fraction=0.7 \
    --learning-starts=10000 \
    --train-frequency=4 \
    --target-network-frequency=1000
```

The original provided hyperparameter doesn't perform well, so I adjust some of them.

- buffer-size=80000 Bigger buffer-size will provide more experience for the model to learn from history.
- gamma=0.99 Since our task is a long-thread task, it's wise to set gamma as bigger as possible to fully learn the method at the first.
- tau=0.005 The parameter added to implement the soft update.
- start-e=1.0 end-e=0.01 fraction=0.7 To make epsilon bigger at first and let it decrease slowlier, that makes the model be more creative in a long period of time.
- train-frequency=4 Make our model learn more frequent, which may helps the model fully learn the experience from the batch.

2.3 Curve Plots

We use TensorBoard to record our training process, the plots are as follows:

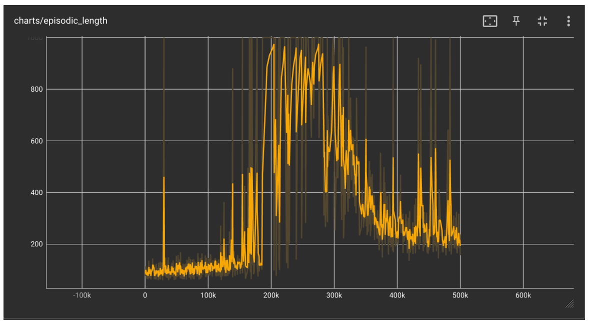

Figure 6: Episodic_length

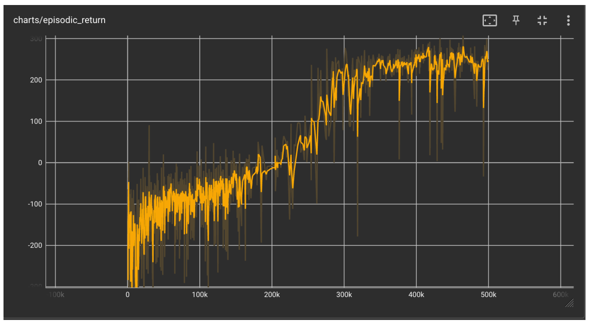

Figure 7: Episodic_return

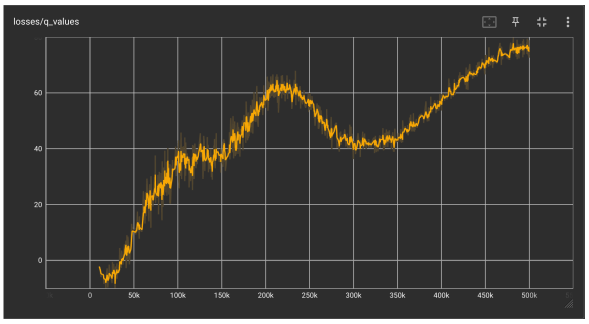

Figure 8: Q_values

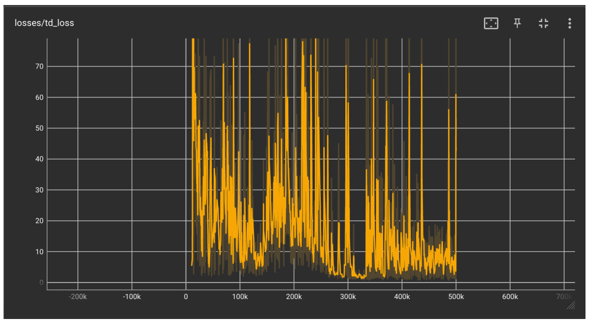

Figure 9: Td_loss

From Figure 6, we can see the average steps our model takes to complete the game rise at the front-mid period, and finally falls up to around 200, which means after many bad trials it forms a relatively stable method to complete it.

The same can be seen from Figure 7, the average return point we achieve keeps grow higher and reach 200 to be stable.

The Figure 8&9 don't perform so well at all. However we can see q_value rises up in the whole trend, though twisting its way. And out td_loss still have some "spike" point at the end of our training process, which may lead to the unstablity of the model.

2.4 Videos

The final result is shown in *dqn.mp4* in *./videos*.

3 About UCB

For the extra learning of exploration schema, I select to learn more about UCB (Upper Coffidence Bound).

Referring articles:
- https://blog.csdn.net/songyunli1111/article/details/83384738
- https://blog.csdn.net/songyunli1111/article/details/83384738
- https://blog.csdn.net/2301_80079642/article/details/148865047

The UCB method is as what its name tells: the model always have confidence about the possiblity of future, and choose the biggest possible reward.

The typical formula of UCB:

$$a = argmax_a(Q(s,a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}})$$

The view should be casted on the latter factor, which is the soul of UCB. We use the overall trial times to divide the trial times that we have spent on this next-status, to show about the ”explored-degree”. This degree is bigger and we consider it has more chance to lead us to a bigger profit, so we are more possible to try it. While smaller means we have try many times on it so we consider it to be not proper for the future planning.

In many long-thread condition, UCB method is proved to be mroe efficient due to its ”braverage” to explore the unknown status, considering ”unknown” to be ”better return”. It’s quite like gambler’s mind, and it is indeed typically aplied to solve some gambling problem like ”Slot-Machine Problem”.

4 Conclusion

By finishing this homework, I really learn a lesson in praticing the agent-training process, it is actually more meaningful to do it myself than just reading documents and ppts.

And I once faced with many problems while adjusting the hyperparameters, and be not confident with my result. Thanks to the help of TAs, they give me advice. Great thanks to my TAs!