AI 3603 Artificial Intelligence: Principles and Techniques

By: LiuQiaoan (520030910220)

HW#: 2

November 15, 2022

I. INTRODUCTION

In AI3603 class, we have studied RL, from model-based learning, such as MDP, to model-free learning, such as Q-learning, SARSA and some other algorithms. In this homework, I try to implement Q-learning and SARSA, and try to tune the hyperparameters of a DQN.

II. REINFORCEMENT LEARNING IN CLIFF-WALKING ENVIRONMENT

In this section, I first complete `cliff_walk_sarsa.py` and `SarsaAgent` in `agent.py`, then I re-use the code to complete `cliff_walk_qlearning.py` and `QLearningAgent` in `agent.py`.

A. SARSA

1. *SarsaAgent*

I try to implement `SarsaAgent`.

```python
class SarsaAgent(object):
    ##### START CODING HERE #####
    def __init__(self, all_actions, epsilon, alpha, gamma):
        """initialize the agent. Maybe more function inputs are needed."""
        # all_actions is a numpy array of all the actions, np.array([0,1,2,3])
        self.all_actions = all_actions
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.Q_s_a = {} # self.Q_s_a is a dict with the form of key as a tuple of s and a, i.e. key: (state, action
            )

    def get_Q(self,state, action):
        self.Q_s_a.setdefault((state, action), 0)
        return self.Q_s_a[(state, action)]

    def choose_action(self, observation):
        """choose action with epsilon-greedy algorithm."""
        # The observation is simply the current position encoded as flattened index. (From https://www.gymlibrary.
            dev/environments/toy_text/cliff_walking/)
        if np.random.random() < self.epsilon:
            action = np.random.choice(self.all_actions)
        else:
            Q = [self.get_Q(observation, action) for action in self.all_actions]
            action = self.all_actions[np.argmax(Q)]
        return action

    def learn(self, state, action, next_state, next_action, reward):
        """learn from experience"""
        # next_action = self.choose_action(next_state)
        next_Q_s_a = self.get_Q(next_state, next_action)
        self.Q_s_a[(state,action)] = (1-self.alpha)*self.get_Q(state, action) + self.alpha * (reward+self.gamma*
            next_Q_s_a)

    def your_function(self, params):
        """You can add other functions as you wish."""
        return None
```

First, I fill `__init__` function, which stores the value of ϵ, α, γ, then initilize $Q(s, a)$ as an empty dict.

Second, I implement a `get_Q` function, which can calculate $Q(s, a)$ from current state and the action to take. The function use the `setdefault` method of dict, which can return the value of corresponding key if key has already been in the dict, otherwise the dict will construct a new key-value pair whose value is 0. Then we simply return $Q(s, a)$.

Third, I fill `choose_action` function, which takes `observation`, i.e. current state, as parameter. We will take a random action with a probability ϵ, or take an action with the max $Q(s, a)$ with a probability $1 - \epsilon$.

Finally, I fill the `learn` function, which takes `state`, `action`, `next_state`, `next_action` and `reward` as parameters. This implementation uses the **true** action a' taken in next state s' to update the Q-value of this state.

2. *cliff_walk_sarsa.py*

Then I try to implement `cliff_walk_sarsa.py`.

```python
####### START CODING HERE #######

# construct the intelligent agent.
alpha = 1.
gamma = .9
epsilon = 1.
agent = SarsaAgent(all_actions, alpha, gamma, epsilon)
reward_list = [0]*1000
epsilon_list = [0]*1000
path = []
```

First, I construct the intelligent agent. I need to set the value of α, γ and ϵ for this agent. What's more, I need to initialize a **`reward_list`** with length 1000 to store the value of reward of each episode, also a **`epsilon_list`** to store the value of ϵ of each episode, and a **`path`** to store the final paths found by the intelligent agent after training.

```python
# start training
for episode in range(1000):
    # record the reward in an episode
    episode_reward = 0
    # record epsilon
    epsilon_list[episode] = agent.epsilon
    # reset env
    s = env.reset()
    a = agent.choose_action(s)
    # render env. You can remove all render() to turn off the GUI to accelerate training.
    # env.render()
    # agent interacts with the environment
    for iter in range(500):
        # choose an action
        s_, r, isdone, info = env.step(a)
        a_ = agent.choose_action(s)
        # env.render()
        # update the episode reward
        episode_reward += r
        print(f"{s} {a} {s_} {r} {isdone}")
        # agent learns from experience
        agent.learn(s,a,s_,a_,r)
        s,a = s_,a_
        if isdone:
            time.sleep(0.1)
            break
    print('episode:', episode, 'episode_reward:', episode_reward, 'alpha:', agent.alpha, 'epsilon:', agent.epsilon)
    reward_list[episode] = episode_reward

    agent.alpha -= 0.0009
    agent.epsilon *= 0.99
print('\ntraining over\n')

# close the render window after training.
```

Second, I fill the training process. At each episode, the agent reads it's initial state and iterates to take action and get next state and next action. Then we use the current state s, current action a, next state s' and next action a' to call **`learn`** function and update $Q(s, a)$. After each episode, the value of α will reduce 0.0009, which ensures the importance of previous samples to be higher with time going by, and the value of ϵ will be 0.99 times it's previous value, which ensures the random exploration will be reduced as time going by. Meanwhile, we need to store the reward and value of ϵ at each episode.

```python
# test
s = env.reset()
path.append(np.unravel_index(s,(4,12)))
agent.epsilon = 0
reward = 0

record_video = 0 # use a flag to set whether recording a video, you can modify it.
if record_video:
    env = gym.wrappers.RecordVideo(env, video_folder = "./video", name_prefix = "sarsa")
else:
    env.render()
while True:
    a = agent.choose_action(s)
    s_, r, isdone, info = env.step(a)
    reward += r
    if not record_video:
        env.render()
    s = s_
    path.append(np.unravel_index(s,(4,12)))
    if isdone:
        break
env.close()
print(f"reward: {reward}")
```

Third, I reuse the code from training to test the agent. I set the value of ϵ to be 0, and get the position of agent using `numpy.unravel_index` (this is because the state of agent is represented as flattened index, then this method can return the un-flattened position). We can also change the value of `record_video` as a flag to decide whether to record a video or not. After each action, we store the state to `path`, and add the immediate reward to `reward`.

```python
# plot
## 1. reward list
if not os.path.exists("./output"):
    os.mkdir("./output")
plt.figure()
plt.plot(reward_list)
plt.savefig("./output/cliff_walk_sarsa_reward.pdf")

## 2. epsilon list
plt.figure()
plt.plot(epsilon_list)
plt.savefig("./output/cliff_walk_sarsa_epsilon.pdf")

## 3. path
print(path)
path = np.array(path)
plt.figure(figsize=(6,3))

plt.xticks([i for i in range(12)])
plt.yticks([i for i in range(4)])
plt.ylim(-0.5,3.5)
ax = plt.gca()                              # Get the current axis information
ax.xaxis.set_ticks_position('top')   # Change x-axis to up
ax.invert_yaxis()                          # Invert y-axis
ax.set_aspect('equal', adjustable='box')

plt.plot(path[:,1],path[:,0])
plt.plot(path[0][1],path[0][0],"g^")
plt.plot(path[-1][1],path[-1][0],"r*")
plt.title(f"Reward = {reward}")
plt.savefig("./output/cliff_walk_sarsa_path.pdf")

####### END CODING HERE #######
```

Finally, I use `matplotlib.pyplot` to plot the episode reward and ϵ value during the training process, and visualize the final path found by the intelligent agent after training.

B. Q-learning

1. *QLearningAgent*

I try to implement `QLearningAgent`.

```python
class QLearningAgent(object):
    ##### START CODING HERE #####
    def __init__(self, all_actions, epsilon, alpha, gamma):
        """initialize the agent. Maybe more function inputs are needed."""
        # all_actions is a numpy array of all the actions, np.array([0,1,2,3])
        self.all_actions = all_actions
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.Q_s_a = {} # self.Q_s_a is a dict with the form of key as a tuple of s and a, i.e. key: (state, action
            )

    def get_Q(self,state, action):
        self.Q_s_a.setdefault((state, action), 0)
        return self.Q_s_a[(state, action)]

    def choose_action(self, observation):
        """choose action with epsilon-greedy algorithm."""
        # The observation is simply the current position encoded as flattened index. (From https://www.gymlibrary.
            dev/environments/toy_text/cliff_walking/)
        if np.random.random() < self.epsilon:
            action = np.random.choice(self.all_actions)
        else:
           Q = [self.get_Q(observation, action) for action in self.all_actions]
            action = self.all_actions[np.argmax(Q)]
        return action

    def learn(self, state, action, next_state, reward):
        """learn from experience"""
        # next_action = self.choose_action(next_state)
        max_next_Q_s_a = max(self.get_Q(next_state, action) for action in self.all_actions)
        self.Q_s_a[(state,action)] = (1-self.alpha)*self.get_Q(state, action) + self.alpha * (reward+self.gamma*
            max_next_Q_s_a)

    def your_function(self, params):
        """You can add other functions as you wish."""
        return None
```

Except for `learn`, other methods are the same as `SarsaAgent`. Since Q-learning is different from SARSA, which uses the max Q-value of next state s' and next action a' to update $Q(s,a)$.

2. *cliff_walk_qlearning.py*

The only difference of `cliff_walk_qlearning.py` and `cliff_walk_sarsa.py` is the training process.

```
# start training
for episode in range(1000):
    # record the reward in an episode
    episode_reward = 0
    # record epsilon
    epsilon_list[episode] = agent.epsilon
    # reset env
    s = env.reset()
    # render env. You can remove all render() to turn off the GUI to accelerate training.
    # env.render()
    # agent interacts with the environment
    for iter in range(500):
        # choose an action
        a = agent.choose_action(s)
        s_, r, isdone, info = env.step(a)
        # env.render()
        # update the episode reward
        episode_reward += r
        print(f"{s} {a} {s_} {r} {isdone}")
        # agent learns from experience
        agent.learn(s,a,s_,r)
        s = s_
        if isdone:
            time.sleep(0.1)
            break
    print('episode:', episode, 'episode_reward:', episode_reward, 'alpha:', agent.alpha, 'epsilon:', agent.epsilon)
    reward_list[episode] = episode_reward
    agent.alpha -= 0.0009
    agent.epsilon *= 0.99
print('\ntraining over\n')

# close the render window after training.
```

We can see that at each time step, agent first use current state s to choose current action a, then agent step by this action to reach next state s'. Now we pass the current state s, current action a and next state s' to `learn`, then Q-learning will choose the next action a' with max value of $Q(s',a')$.

Then we use the same code to test and plot the episode reward and ϵ value during the training process, and visualize the final path found by the intelligent agent after training.

C. Result and Analysis

1. *Analysis of learning process*

Learning process of SARSA.

1. First, initialize state s, and take action a from current state with ϵ-greedy.
2. Second, for each step of episode, we do the following things.
    - Take current action a to observe the immediate reward r and next state s'.
    - Choose next action a' from s' with ϵ-greedy.
    - Update $Q(s,a)$ with $(1-\alpha)Q(s,a) + \alpha\,[r + \gamma Q(s',a')]$.
    - Set current state with s' and current action with a'.
    - Until current state is the goal state, the episode is over.

Learning process of Q-learning.

1. First, initialize state s.
2. Second, for each step of episode, we do the following things.
    - Take current action a to observe the immediate reward r and next state s'.
    - Calculate $Q(s', a')$ for all next action a' and get $max_{a'}Q(s', a')$.
    - Update $Q(s, a)$ with $(1 - \alpha)Q(s, a) + \alpha\,[r + \gamma max_{a'}Q(s', a')]$.
    - Set current state with s'.
    - Until current state is the goal state, the episode is over.

When running the script to training SARSA model, I find that it takes longer time to converge, and from the reward curve of SARSA in Figure 1, we can see that reward oscillately go up and finally converge to -17. The learning process of SARSA has more **oscillation**.

Q-learning model really takes less time to converge, and the reward curve seems to be more steady, seen in Figure 4, which converge to -13.

2. Difference

We can see from Figure 3 and Figure 6 that SARSA may learn a path as far away from cliff as possible, however, Q-learning seems bravely pass nearby the cliff to take the shortest path to the end. The difference is because that SARSA is **on-policy learning**, while Q-learning is **off-policy learning**.

Sarsa takes a conservative approach, updating its $Q(s, a)$ with its actions really taken in the next time, so it is sensitive to errors and death, like a cautious man, Q-learning, on the other hand, just take the max $Q(s', a')$ every time it wants to take next action, no matter how dangerous this way may be. So Q-learning is a reckless, bold, and greedy algorithm that does not care about death and mistakes.

Therefore, the difference between on-policy learning and off-policy learning causes the difference in paths. SARSA wants to take a safe way to the goal, thus its path will be away from the cliff and converges slower. Q-learning wants to take risks to find the shortest way no matter the danger along the way, so it dances on the edge of cliff and converges faster.

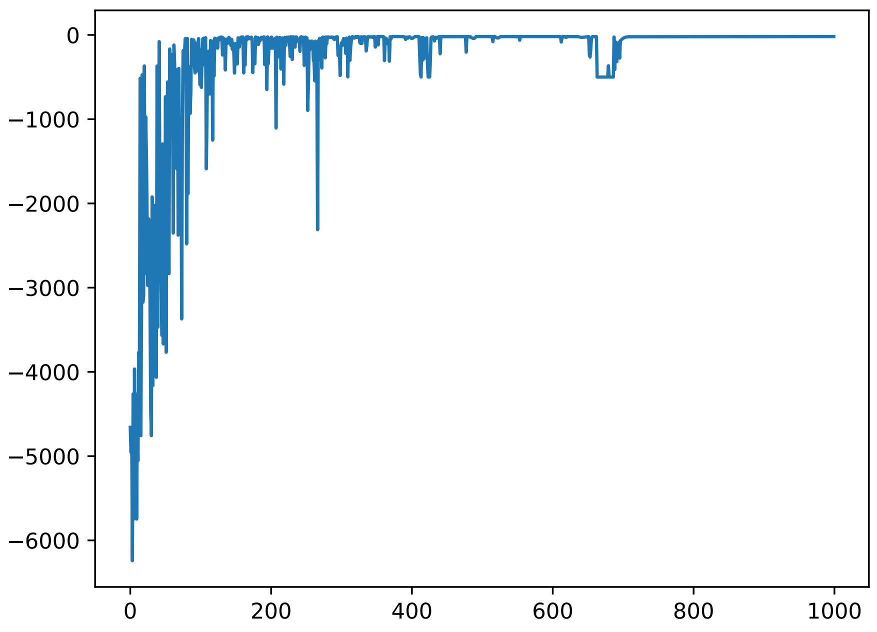

FIG. 1: The episode reward of SARSA during the training process.

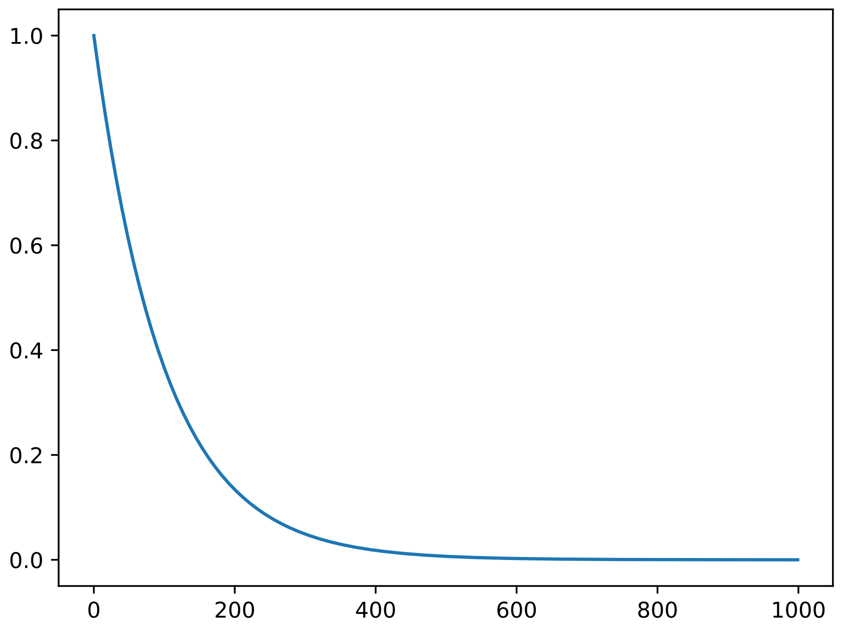

FIG. 2: The ϵ value of SARSA during the training process.

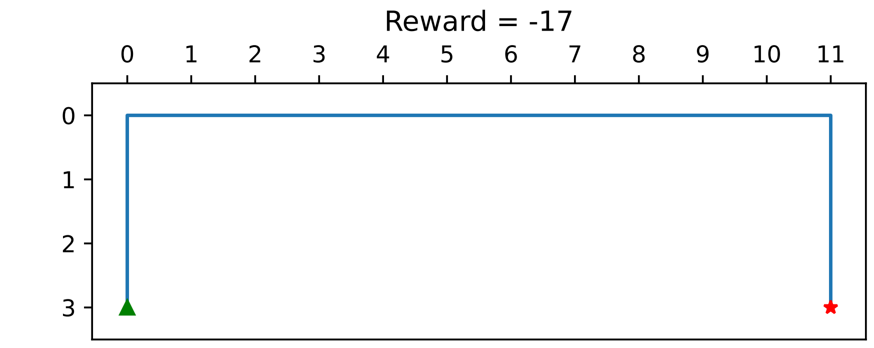

FIG. 3: The final paths found by the intelligent SARSA agent after training. The green triangle stands for start point and the red star stands for final point.

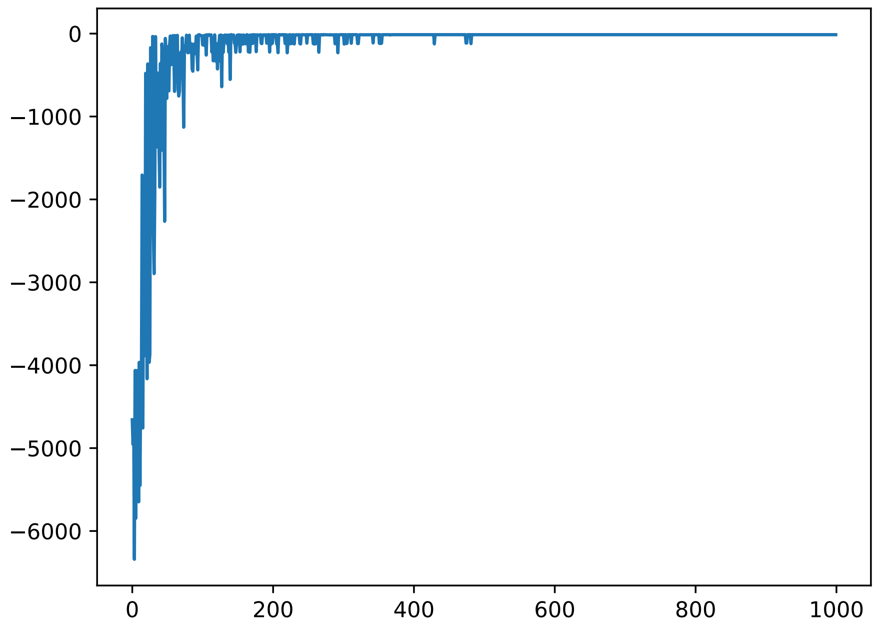

FIG. 4: The episode reward of Q-learning during the training process.

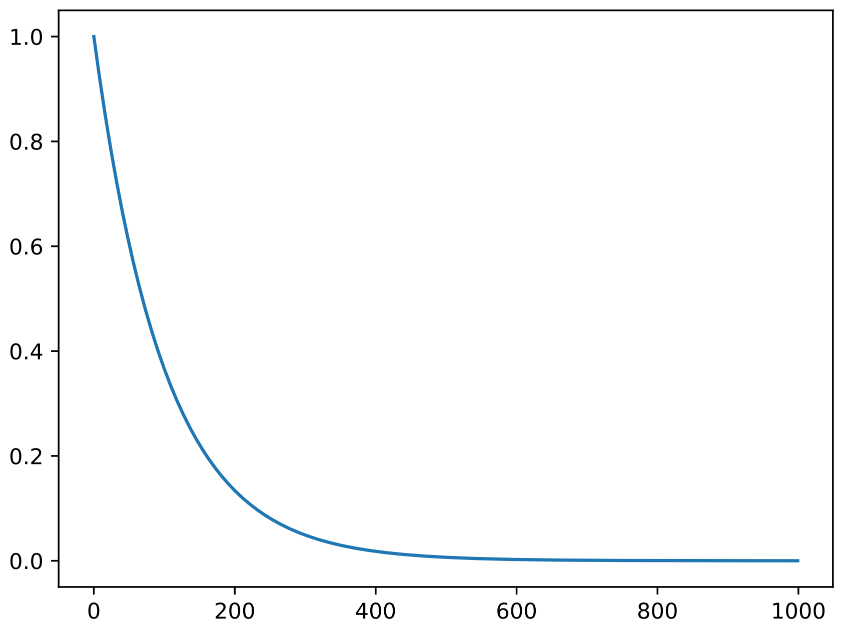

FIG. 5: The ϵ value of Q-learning during the training process.

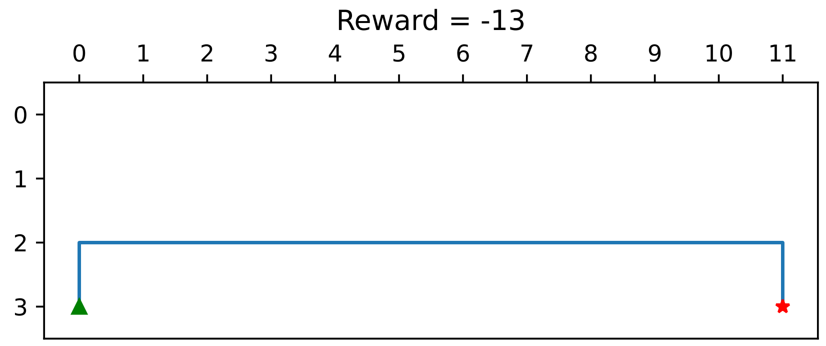

FIG. 6: The final paths found by the intelligent Q-learning agent after training. The green triangle stands for start point and the red star stands for final point.

III. DEEP REINFORCEMENT LEARNING

A. Read and Analysis

First, I read the code and fill all the ”comments” blocks.

B. Train and Tune the Agent

Second, I train and tune the agent.

If I use the initial given parameters, however, the agent seems to thrash a lot, and the reward curve seems to be not converged, see in Figure 7

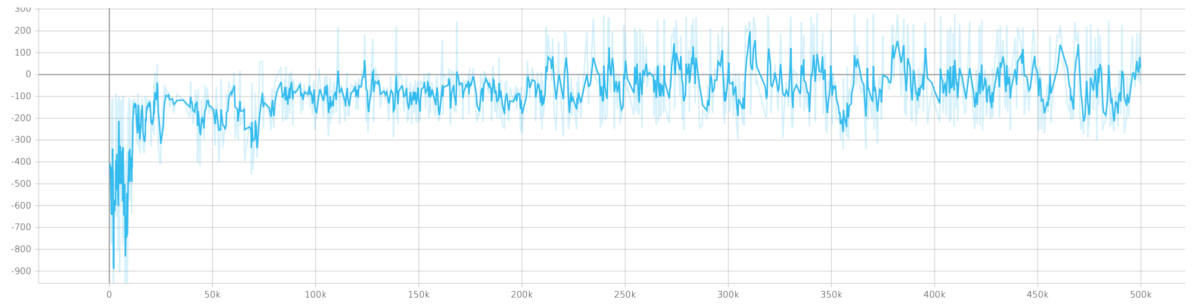

FIG. 7: The curve of episode return of the initial agent.

I first tune the value of γ. I find that if I set γ as a relatively small value, e.g. $\gamma = 0.5$, it seems that the agent will hover in the sky but not reach the land. I think this is mainly because the final positive reward will be so small if we use small γ, then the agent will not take risks to land, bacause it will probably get worse reward if it really lands. So I tune γ to 0.99, then the agent can be land on the ground but it has some thrashing.

Then I go to change the value of start ϵ and end ϵ. At the beginning of training, I want the agent to have more freedom to explore the environment, so I set the start ϵ to be 1.0. And at the end of training, I want the episode return to be converged, so I set the end ϵ to be 0.01, which is a small value. After I training the agent, it seems to have more stability, but take a long time to learn, and the action seems so cautious and slow.

Therefore, I change the value of learning rateto be 0.0025 because I want a quicker training process and does not want the model to be overfitting. I find that in this time, the agent perform really good and the episode reward nearly converges to about 270. We can see the curve of episode return at Figure 8.

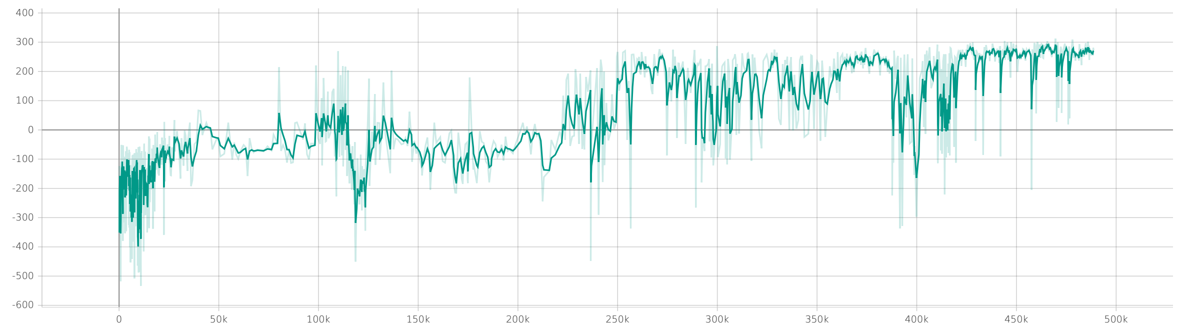

FIG. 8: The curve of episode return of the final agent.

C. Improve DQN Agent

My attention mainly focus on the **training efficiency** of DQN. As we use Replay Buffer and two network, the time consumed in training is quite long. Then I find that this paper [1] has proposed an effcient algorithm call A3C to solve the drawback of DQN.

First, A3C is an asynchronous variant of Actor-Critic algorithm, which combines the advantage of SARSA, Q-learning and policy gradient.

Then, A3C uses multi-threading strategy, it sets up several agents with the same structure in different threads. Then the agents train parallelly, and update the parameters of the target network. With this algorithm, the parallelism of agents not only speed up the training process, but it can ensure that agents don't interact with each other. What's more, the discontinuity of training process of different agents also make the correlation of updates to target network become lower, thus improve the robustness.

IV. CONCLUSION

In this homework, I implement two algorithms: SARSA and Q-learning. They are the basis of reinforcement learning, and have quite good performance in toy games. What's more, DQN is also a promising model, which combines the advantage of RL and DL, then it can learn more high-dim features.

In my perspective, though DL has prevailed in the past decade, we have many chances to make it embrace more fields, such as RL, transformer, motion control and so on. And RL is still young, it has many oppitunities to go further.

[1] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.