CSYE 7370 Final Project

Optimization Of Self-driving Car Based On Velocity Adjustment in Reinforcement Learning

Introduction

Overview

Self driving cars are quite popular these days and a higher level of automation in terms of driving has the potential to reduce dangerous drivers behaviors, which may improve road safety and pedestrian safety. Therefore, we think it would be an important and interesting topic to do research on. Our setup is: the agent is driving on a straight road, while several pedestrians are standing on the pavement and there are multiple traffic signs and traffic lights along the way. Our goal is to train the agent such that it slows down when approaching the pedestrians and follows all the traffic rules. In addition, our agent still has a certain velocity when it reaches the destination.

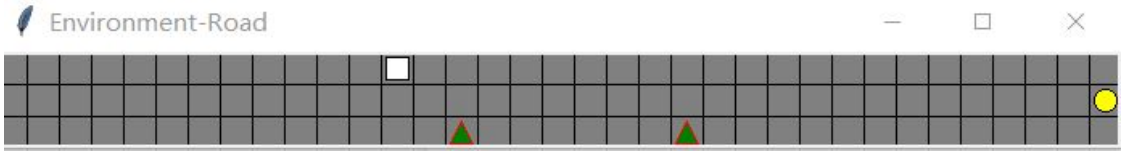
Reinforcement learning is considered a powerful AI paradigm that can teach machines by interacting with the environment and learning from mistakes. Despite its practicality, it has not been successfully used in automotive applications. Inspired by Google DeepMind's successful demonstration of Atari games and Go learning, we proposed a framework for using reinforcement learning for autonomous driving. This is particularly relevant because it is difficult to treat autonomous driving as a supervised learning problem due to the strong interaction with the environment (including other vehicles, pedestrians, and road works). Therefore, we will focus on whether we can use some algorithms of reinforcement learning to better realize the reasonable planning of the speed when the autonomous car interacts with some environments or people.

Scenarios

1. The self-driving vehicle is driving on a straight road, but there are 2 pedestrians standing on the pavement near the driving road, the vehicle should slow the velocity when passing by these 2 people.
2. The self-driving vehicle should arrive at the termination of the road as fast as possible in the condition of safely passing by pedestrians.
3. The self-driving vehicle needs to pass the end point at a prescribed speed

Visualization

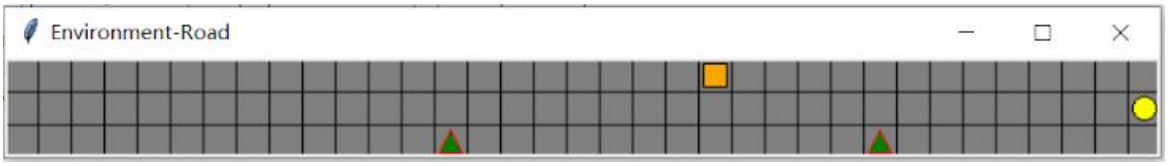
- The grey grid: Roads and Pavements
- The square: The vehicle/agent
- Different colors of square: Different velocities of the vehicle/agent
- Yellow circle: The destination of the road
- Green triangle: Pedestrians



Github Link

https://github.com/pandaYixuan/CSYE7370Final-Self-driving-vehicle-RL

Environment



We use Tkinter to build a simple car driving environment. The entire environment is a 3*35 gray grid, where the first row represents the road where the car is driving, and a square that changes color as the speed changes is used to represent the vehicle, and the vehicle travels from left to right. The speed of the vehicle is represented by moving several squares at a time. For example, when the speed is 2, the vehicle will move two squares to the right in the next step. The initial speed is 3, the faster the vehicle speed, the darker its color. The vehicle has 5 actions [maintain speed, speed +1, speed +2, speed -1, speed -2];

A yellow circle in the second row indicates the end point, and the car passes the end point at the specified speed is a task to complete (which is 4);

There are two red-edged green triangles in the third row, representing two pedestrians. The second task completion is the car not overspeeding past the pedestrian (the maximum speed of the car passing the pedestrian is 2). The two pedestrians randomly change their positions during each experiment to increase the complexity of the environment.

Reward Setting:

1. Effectiveness:
- When the car passes the end point at the specified speed, the reward is increased by 60; otherwise, the reward is increased by -60.
- Each step of the car will increase the reward of -4.
- When the car speed is lower than the minimum speed (0), the reward of -16 is increased.

2. Traffic rules:
- When the car has an overspeeding (speed greater than 4) during the entire journey, the reward of -12 is added.

3. Safety:
- When the car is speeding past pedestrians (speed exceeds 2), the reward of -60 is increased.

4. Comfort:
- When the car speed is negative, the reward of -20 is increased. Every time the car changes its behavior, the reward is increased by -3.

Algorithm and Evaluation

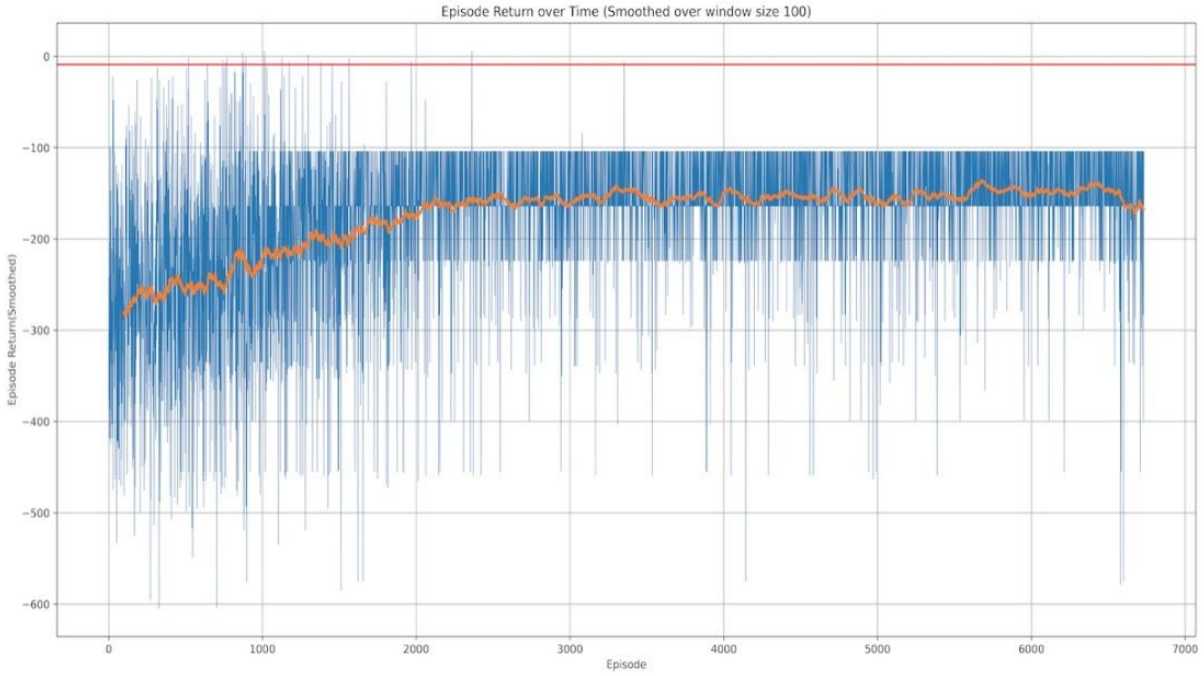
In this project, we tried several types of algorithms for agent learning,
- Monte-Carlo Algorithm
- Q-learning Algorithm
- SARSA Algorithm
- SARS-Lambda Algorithm
- Dynamic Programming Method
- Deep Q-learning Algorithm

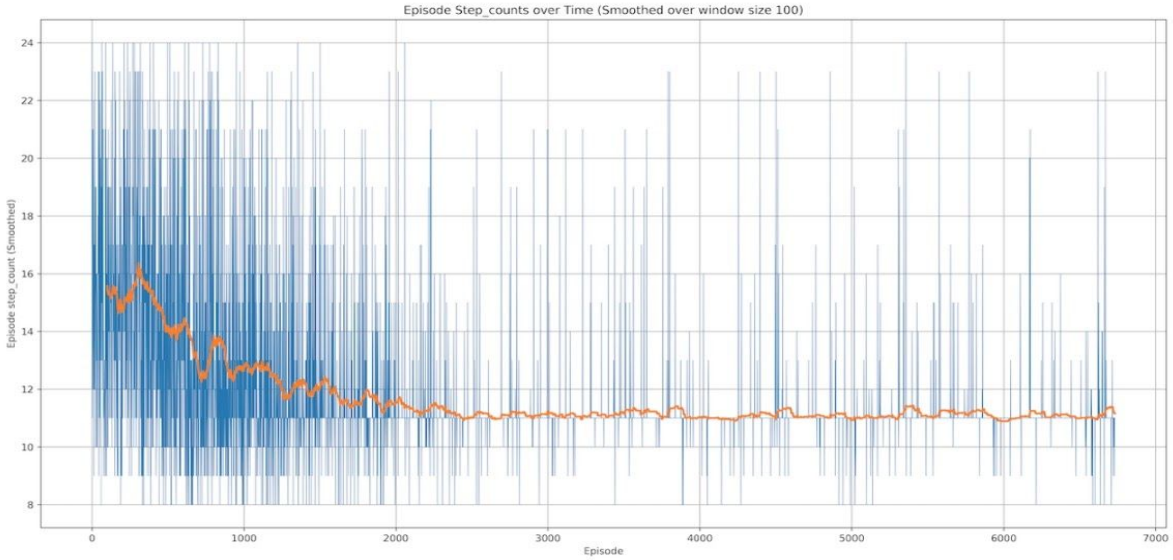
Monte Carlo

Monte Carlo methods are used to model the probability of various outcomes that could be generated from a process, which is not easy to predict because of intervention of random variables. It basically relies on random variables to generate numerical results. It is a method usually used to predict potential impact of risks and uncertainties in prediction.Monte Carlo methods can be used to solve a wide range of problems including engineering, science, finance, games. It can be referred to as a multiple probability simulation as well.

We applied this technique to our problem, and it did not perform as good as q_learning, but we can still see the agent is learning and improving during the process, which means Monte Carlo still works for our environment.

Here are the results generated using Monte Carlo:
The first diagram shows the reward of every episode, after approximately 200 steps, our agent learned what action should be taken given a certain state. But still ran into taking wrong actions that would result in negative rewards or low rewards. The second diagram shows the average steps taken each episode.And it is decreasing in general which shows that our agent is learning and improving and Monte Carlo method is working for our environment.





Q-learning

Q-learning is an off policy reinforcement learning algorithm that aims to find the best action according to the current state. The reason why q-learning is off policy is because it learns from actions which are outside the current policy, at certain point, the agent would take some random actions to proceed, thus a policy is not needed. The ‘Q’ stands for quality which represents how effective an action is in terms of getting future rewards.

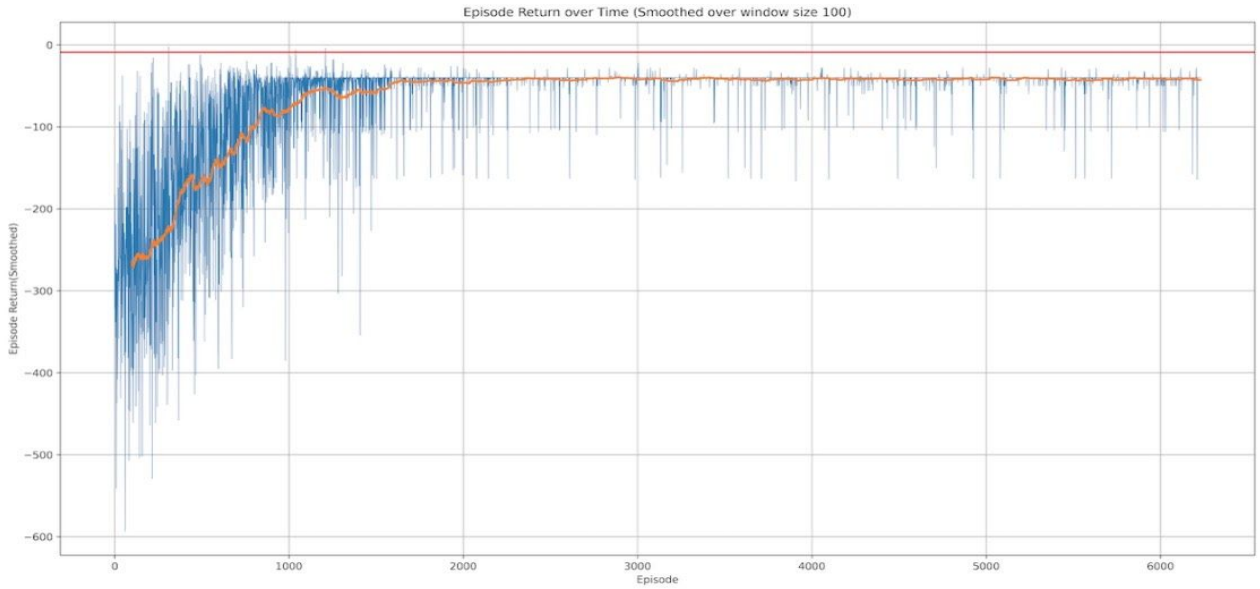
$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

During training, a q-table is created, and it’s size is defined by state*action. The initial values of every q_value in the q-table is equal to zero and then we update and and save the q-values after every episode. And then the agent chooses the optimal action according to the q_values.

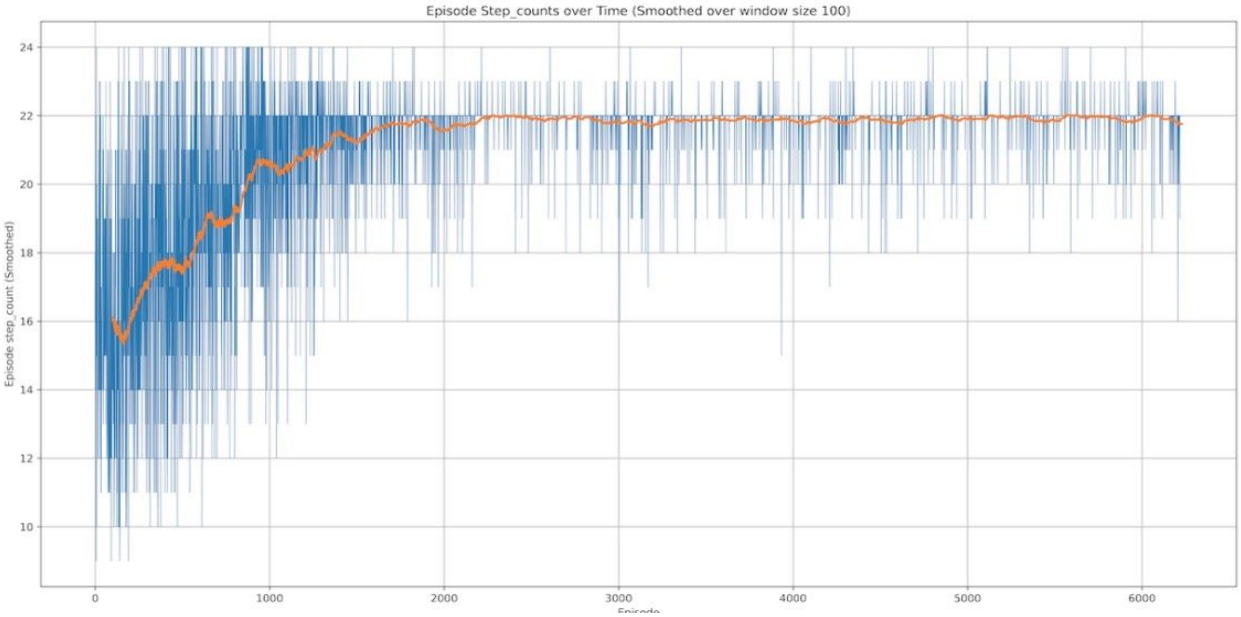
The steps are: the agent starts off in a given state and it takes an action and then receives a reward. Then the agent chooses an action which has the maximum reward or choose to explore by selecting a random action and them update the Q_table

We have five actions in out environment, which are "no_change", "speed_up", "slow_down", "speed_up_up","slow_down_down" with two state "position", "velocity". And we grant rewards to 10 situations: "goal_with_good_velocity": 60, "goal_with_bad_velocity": -60, "per_step_cost": -4, "under_speed": -16, "over_speed": -12, "over_speed_2": -12, "over_speed_near_pedestrian": -60, "negative_speed": -20, "action_change": -3

Here are the results produced by applying q-learning to our environment:



The first diagram shows the reward of every episode, and we can see from approximately after 1500 episodes, the rewards tend to not vary largely which means our agent has found out which action to choose given a state based on q_learning. And the second diagram shows the average steps our agent takes every episode.



SARSA(State–action–reward–state–action):

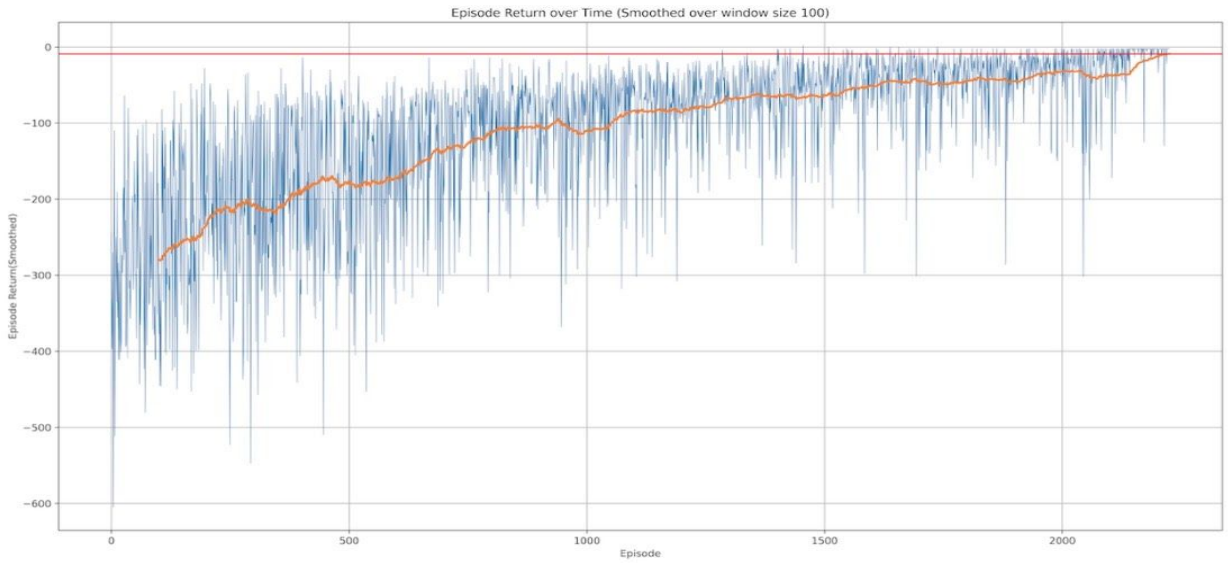
Sarsa is an algorithm which is for learning a process based on Markov decision, which is often implemented in reinforcement learning.

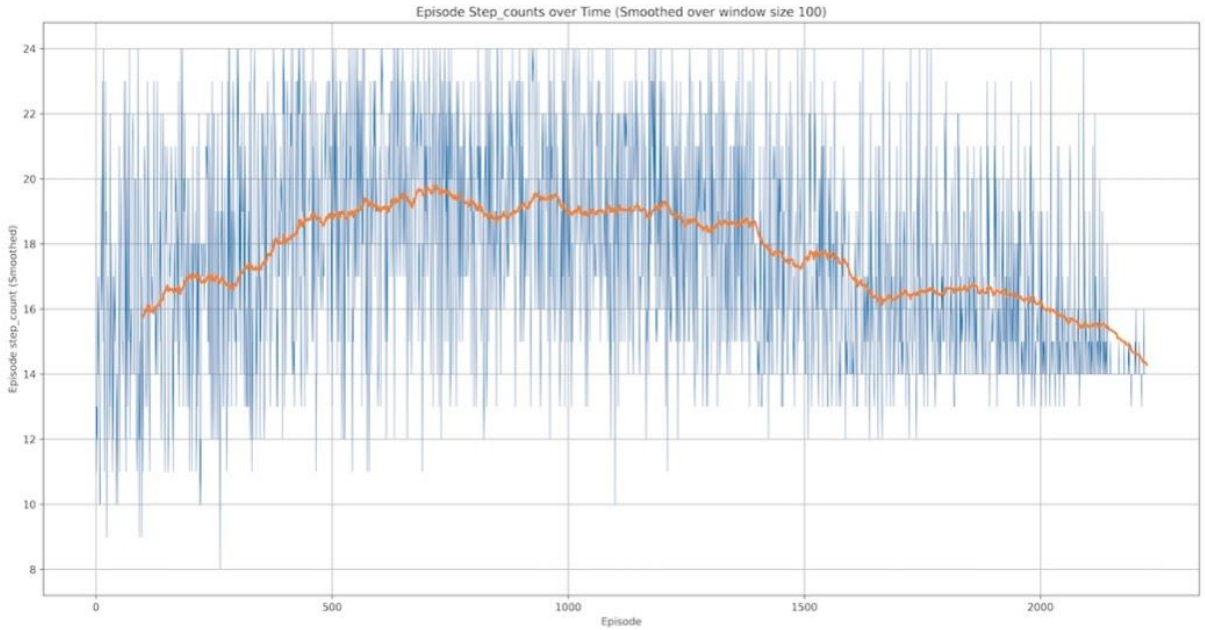
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

Basically an agent interacts with the environment given and updates the policy according to the action that is taken. Therefore, sarsa is considered as a on-policy reinforcement learning algorithm. The q_value is later adjusted by alpha which is the learning rate after being updated.Q value indicates the possible reward that can be obtained in next time stamp by taking a certain action a in a given state s and add the discounted future reward obtained from the observation of next state-action.

Here are the results generated based on SARSA:

The first diagram shows the average reward of each episode. We can see that the line is gradually increasing, which means that our agent is learning. Even though the learning process is not as good as q_learning and monte carlo, eventually, it reaches the optimal cumulated rewards, which also indicates that our agent is improving and sarsa works for our environment. And the second diagram shows the average steps the agent is taken for each episode. As more episodes are trained, the agent takes less steps, which indicates our agent is improving its performance.





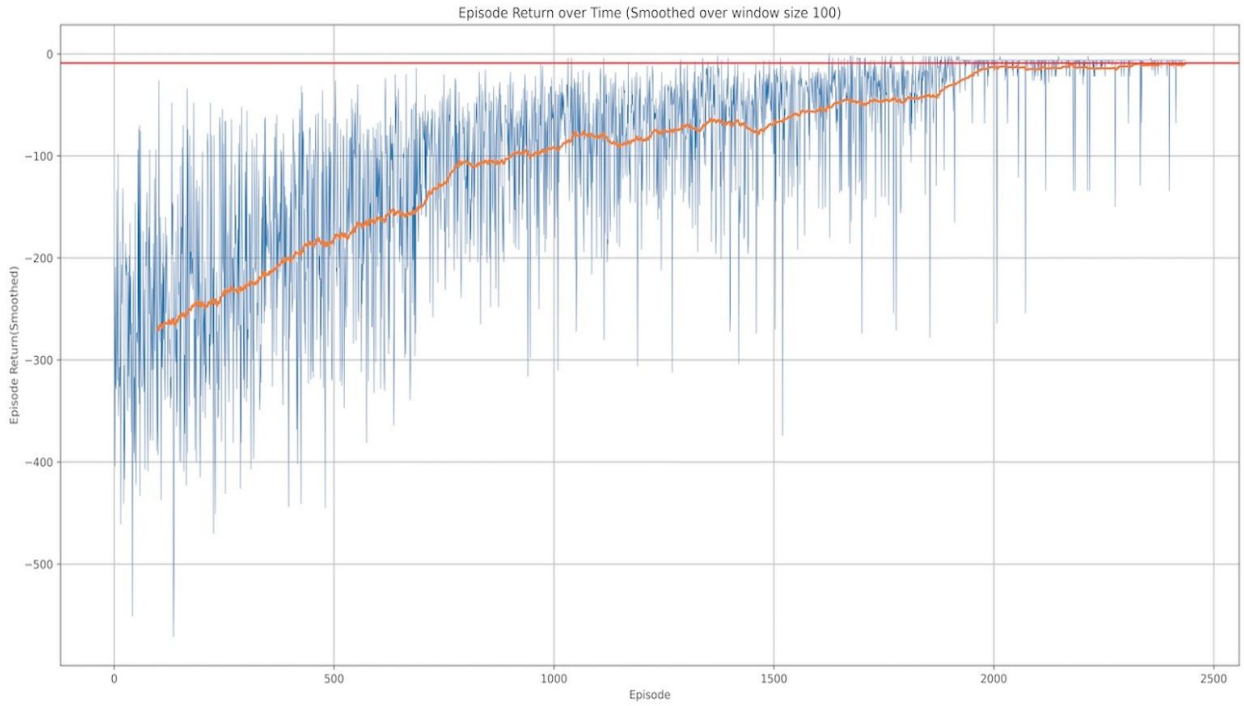
SARSA Lambda:

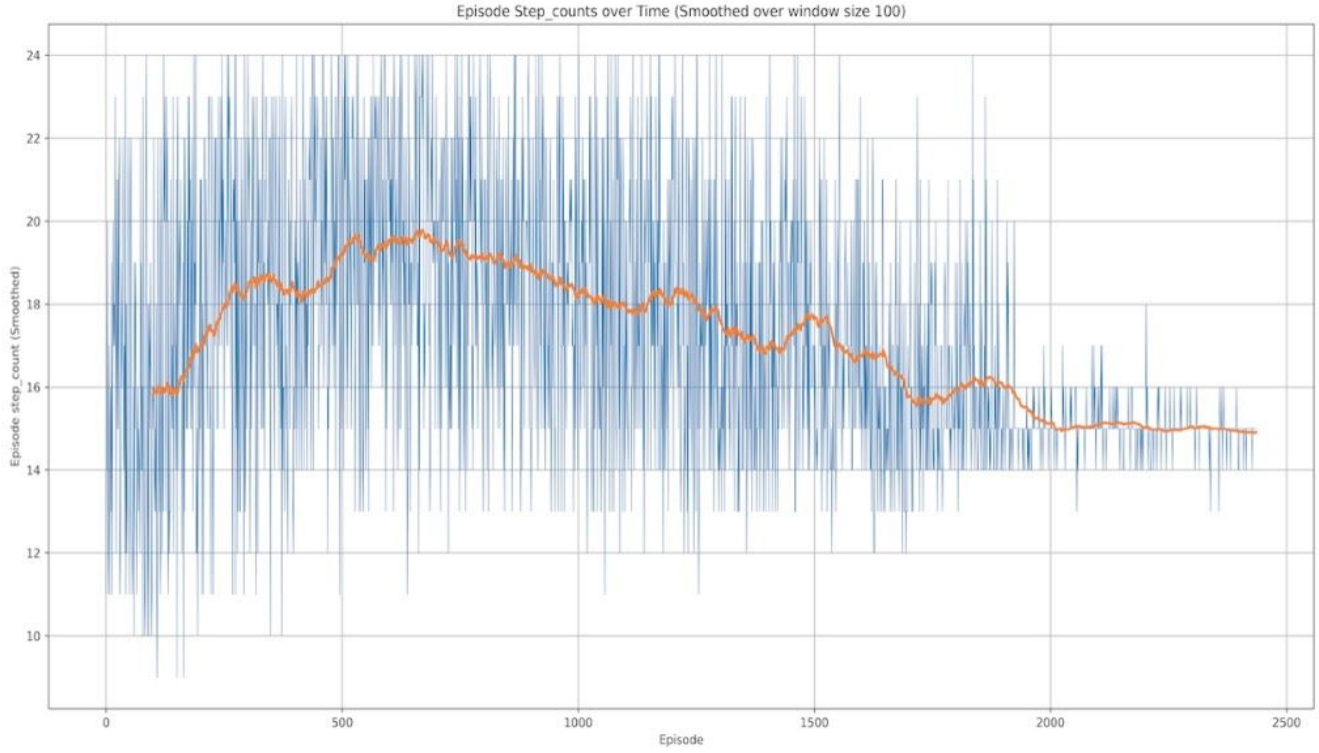
Sarsa Lambda is similar to th Sarsa algorithm with added eligibility traces. It is similar to Sarsa algorithm, except it keeps eligibility trace for each state-action pair.Sarsa Lambda uses traces which can be replaced. And then the algorithm runs a large number of episodes, until the a reasonable accurate value is generated by the action-value function. And here is the pseudocode for this algorithm:

```
Repeat(for each episode)
    Initialize s,a
    Repeat(for each step of epsiode)
        Take action a, observe r, s'
        Choose a' from s' using the beh
        δ ← r + γQ(s',a')-Q(s,a)
        e(s,a) ← 1
        For all s,a:
            Q(s,a) ← Q(s,a) +αδe(s,a)
            e(s,a) ← γλe(s,a)
        s ← s'; a ← a'
```

Here are the result produced by implementing SARSA Lambda to our environment:

The first diagram shows the average reward of each episode. We can see that the line is gradually increasing, which means that our agent is learning. We can see that the learning result is better than the one produced by Sarsa, the agent took less episodes to reach the optimal rewards that it can get, which also indicates that our agent is improving and Sarsa Lambda works for our environment. And the second diagram shows the average steps the agent is taken for each episode. As more episodes are trained, the agent takes less steps, which indicates our agent is improving its performance.


Episode Return over Time (Smoothed over window size 100)
Episode Return(Smoothed)
0
−100
−200
−300
−400
−500
0
500
1000
1500
2000
2500
Episode


Episode Step_counts over Time (Smoothed over window size 100)
Episode step_count (Smoothed)
24
22
20
18
16
14
12
10
0
500
1000
1500
2000
2500
Episode

Dynamic Programming

Dynamic Programming or (DP) is a method for solving complex problems by breaking them down into subproblems. Solving the subproblems, and then combining solutions of the solved subproblems to tackle the overall problem.

This project is Markov Decision Processes (MDPs) which has two really important properties that makes the Dynamic Programming method fit for:

- Optimal substructure: we can solve some overall problem by breaking it down into two or more pieces, solve for each of the pieces, and the optimal solution to the pieces tells us how to get the optimal solution to the overall problem.
- Overlapping subproblems: The subproblems which occur once, will occur again and again. We broke down the problem of getting to the wall into first getting to the midpoint, and then getting to the wall. That will help us solve the subproblem of how to get from another point to the wall. Solutions can be cached and reused. (What gives us the overlapping subproblems property in MDPs is the value function. It's like a cache of the good information we figured out about the MDP)

In the Dynamic Programming methods for this project, it consists of the following parts:

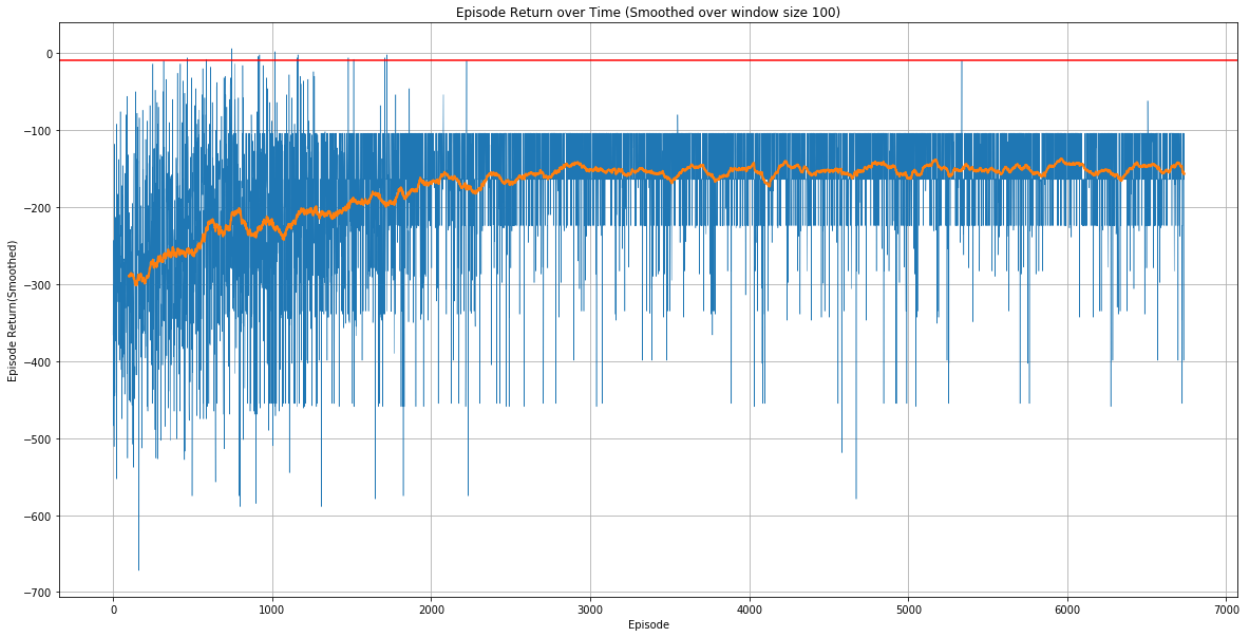
- Policy Running
- Policy Evaluation
- Policy Improvement
- Policy Iteration
- Value Iteration

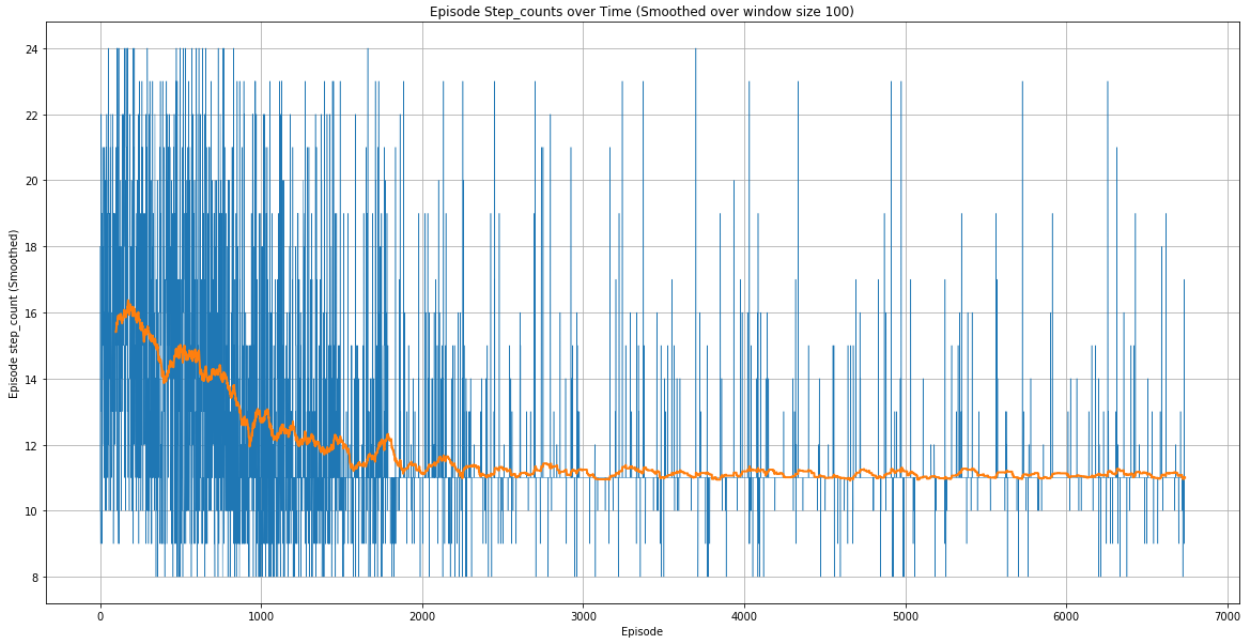
And there are several ways to apply the Dynamic Programming method for optimizing the agent learning.

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

From David Silver class

Bellman equation gives us recursive decomposition (the first property). Bellman equation tells us how to break down the optimal value function into two pieces, the optimal behavior for one step followed by the optimal behavior after that step.



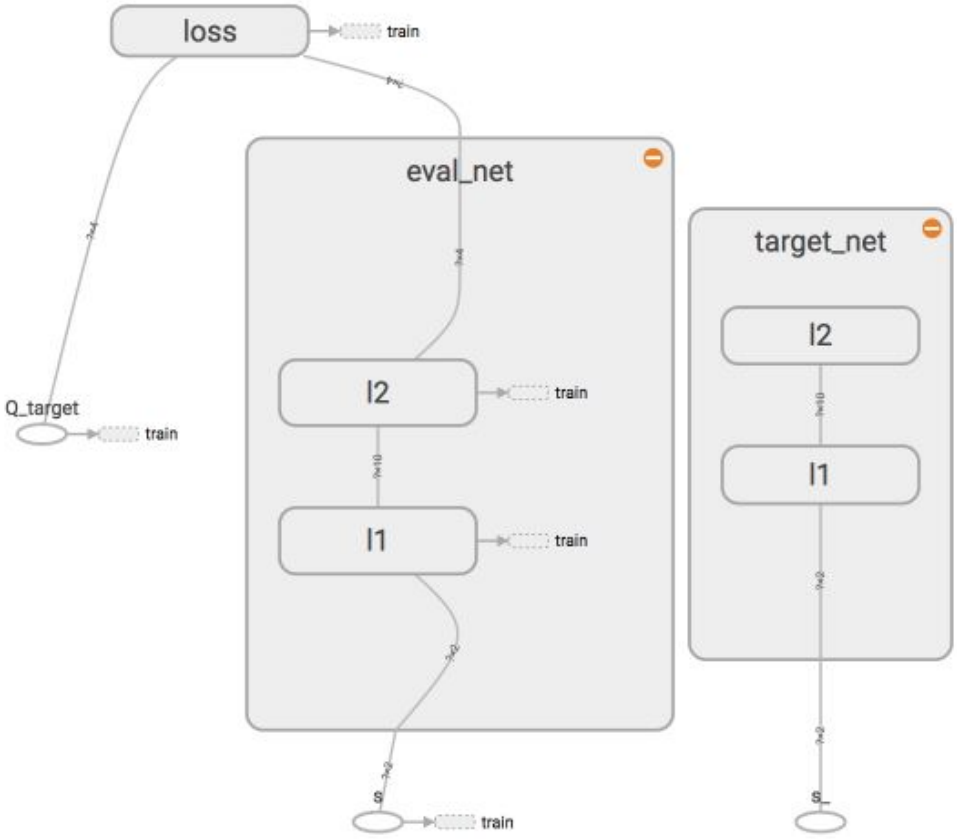


Deep Q-Network

Introduction

Deep Q-Network is a reinforcement learning algorithm that combines Q-Learning with deep neural networks to let RL work for complex, high-dimensional environments.
We implemented DQN for our environment to compare with the Q-Learning algorithm.

Network Graph



Algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

Build Network

We build two networks, target network and evaluate network. Target network is a historical version of evaluate network. It includes parameters from the previous version of the evaluation network. Those parameters will not be changed during some preset time. Then it will be changed by the new parameters from the evaluate network. And the evaluate network is trainable.

Store the transition in memory

Store the transition that the agent obsers, then those observations can be reused for the model to learn.

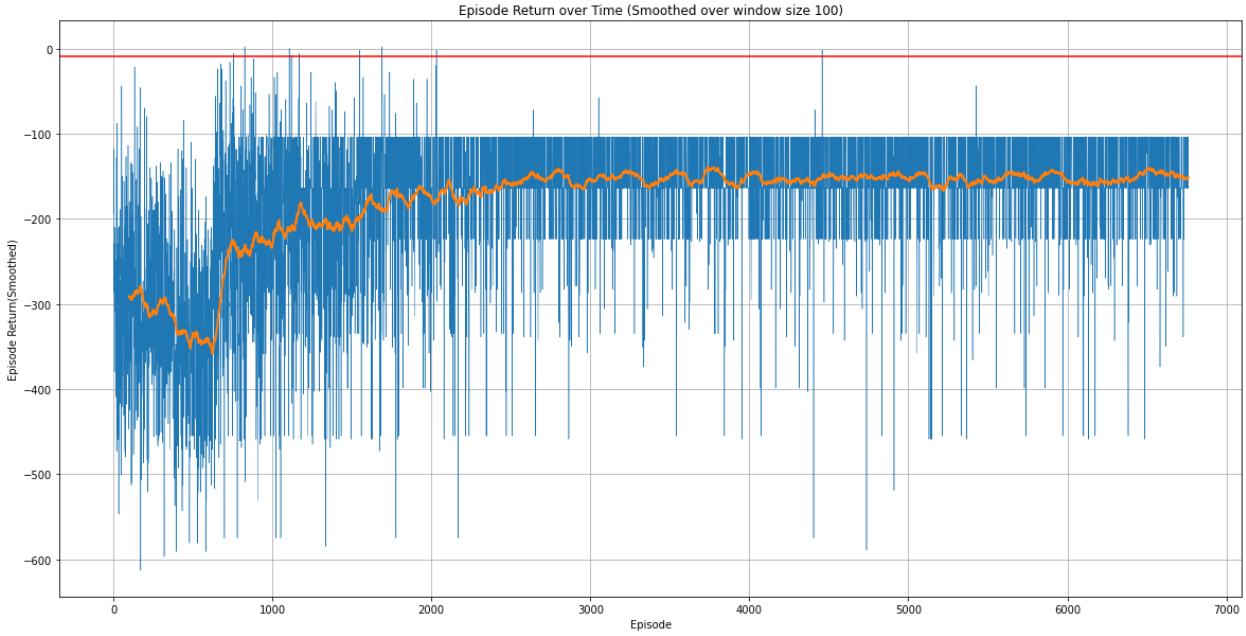
Choose action

Similar to Q learning and Sarsa, DQN will choose action based on the epsilon. It will exploration to choose a random action or exploitation to choose the action that has largest Q value

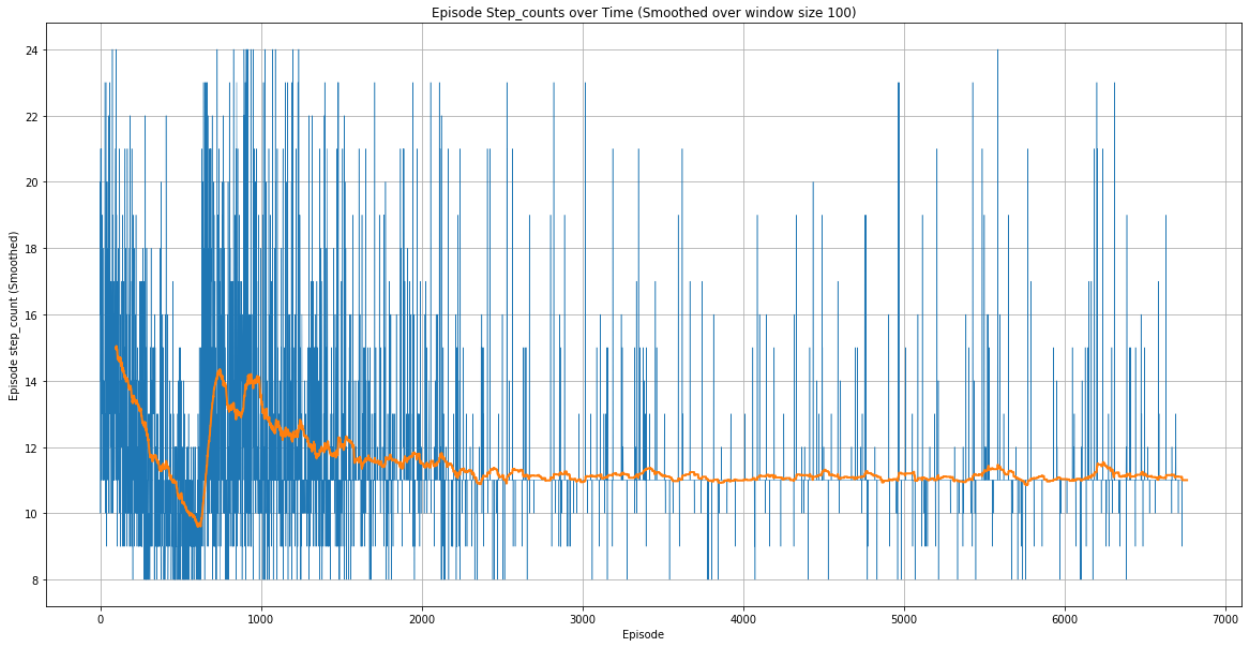
Learning

The learning process will allow the agent to update parameters and keep improvement. In the DQN, we will perform a gradient descent step on the evaluate network.

Result



As we saw in the Episode Return Result. This graph shows the reward per episode. At the beginning, the statistics were decreasing due to the epsilon being greedy and after 800 episodes, the reward kept increasing. So we consider the DQN is working.



However, the biggest problem is that after 2000 episodes, most episodes will be done after less than 11 steps. It limited the result of the DQN model.

Future Works

In general, most of the algorithms have good performance in this simple car speed control environment, they all have amazing learning speed and good learning results. In the next step, we can build more complex and diverse environments to evaluate the performance and learning of different algorithms. At the same time, how to better adjust the parameters and structure of DQL to have better learning results is also the goal that we need to further experiment and explore.

References

https://mofanpy.com/tutorials/machine-learning/reinforcement-learning
https://towardsdatascience.com/reinforcement-learning-demystified-solving-mdps-with-dynamic-programming-b52c8093c919
http://www.incompleteideas.net/book/RLbook2020.pdf
https://arxiv.org/pdf/1704.02532.pdf
https://ieeexplore.ieee.org/abstract/document/8248668
https://github.com/openai/gym
https://towardsdatascience.com/reinforcement-learning-dynamic-programming-2b89da6ea1b

Team Members

Bo Cao 001065055
Haoyu Yin 001875345
Qianjin Wu 001448032
Yixuan Wang 001494410