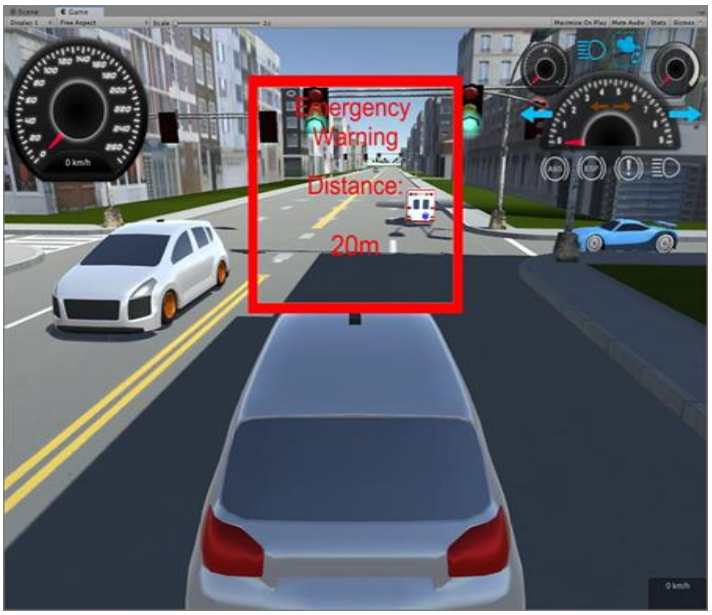
Deep Imitation Learning for Autonomous Driving based on Cyber-Physical System

Xiangyu Li [a], Ding Cao [b], Ivan Wang-Hei Ho [b]

[a] Department of Civil and Environmental Engineering, University of California, Berkeley, United States
[b] Department of Electrical and Electronic Engineering, The Hong Kong Polytechnic University, Hong Kong

Introduction:

This project is based on the cyber-physical system for the vehicular networks and intelligent transportation systems. After implementing the algorithms in the CPS, the driver (player) can react according to different V2X messages, and we can collect a huge amount of driving data through the physical driving simulator. These driving data are invaluable for evaluating the efficiency of the proposed algorithms and models, studying the impact on drivers, as well as training deep learning models for autonomous vehicles that are sensitive to V2X messages. In this project, we set up an emergency vehicle broadcast message scenario based on the digital twin platform as shown in the figure below.



We want to train an automatic driving system that can perfectly adapt to the emergency vehicle situation, and improve the efficiency while ensuring the safety of the overall system, that is, the average waiting time. We collected the standard deviation of speed (SDspeed), standard deviation of heading error (SDHE), mean heading error (meanHE), standard deviation of lateral position (SDLP) data for the simulations on the Logitech G29 driving simulator. We use these data to train the autonomous driving system, optimize these four parameters to ensure the safety of each autonomous vehicle, and ultimately reduce the average waiting time of the entire system.

Project setting:

The data is collected from different types of participants on the driving simulator, and preprocessing was made on the data. After that we used the Simulation of Urban MObility (SUMO) simulator

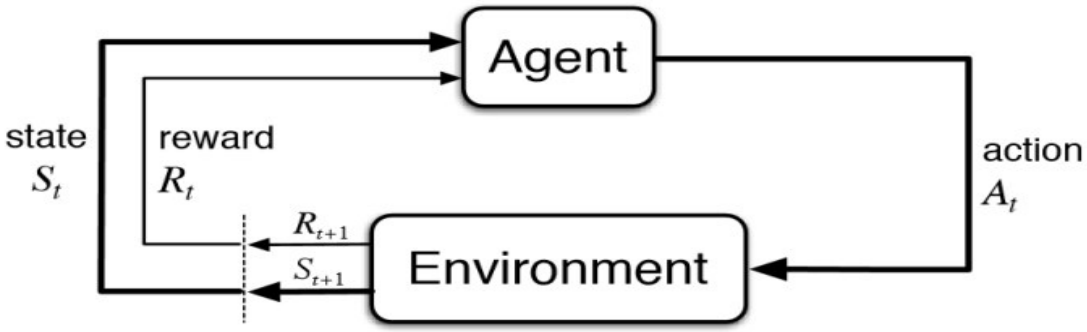
platform, on a windows-based operating system, where I made single agent simulations, and multi agent (2 agents) simulations, which we can see further in this document. The data of cars flow was generated in the SUMO simulator, and was transferred to python with the Traci library.

Single agent case:

In this setting we first create a traffic simulation which only contains a single intersection. The traffic is coming from all sides of the road, and we named each side of the road as North, East, West

and South. Each road has three options to cross the intersection – Going straight, turning left or right. For traffic generation we use Simulation of Urban Mobility (SUMO), as shown in the picture below.

WHAT IS Reinforcement Learning?



Reinforcement Learning is a branch of Machine Learning with a specific structure and flow, as described in the chart. An RL agent performs an action in an environment, which causes some kind of reaction/result in the system. The result that appeared yields a reward – whether the consequences were good or bad. These results are being transferred back to the agent, which adapts itself corresponding to the reward it got from his last action.

RL algorithms can be divided into two types: model-free RL algorithms and model-based RL algorithms. In model-free RL algorithms, we do not have an exact model of the environment, which means that we do not know what will happen in the next time step after we performs an action. On the other hand, in model-based RL algorithms, agent must have to learn the model of the environment which may not be available in most of the real-world problems.

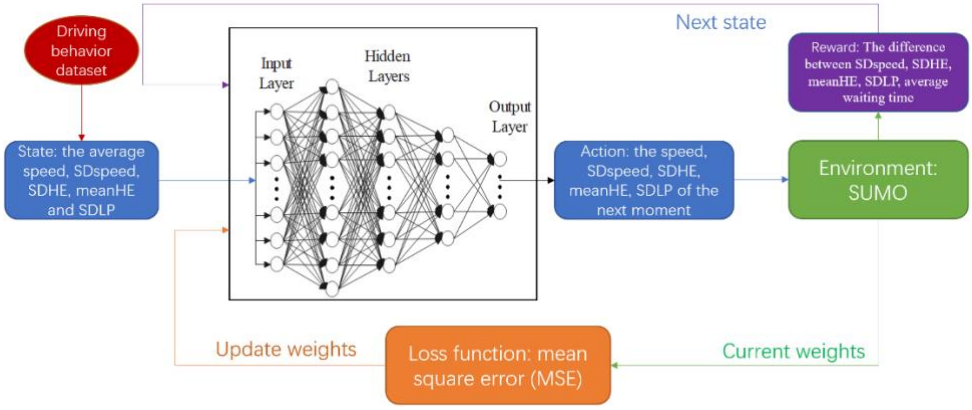
In this work I used model-free RL algorithms to control the overall automatic driving system.
Model-free algorithms further can be classified as **value-based (Q-learning and QL with Neural Networks = DQN) and policy gradient (A2C, PPO)** algorithms.
In this work we will work with both value based and policy gradient algorithms, and examine their success on our problem.

Since we didn't learn these in class, I'll elaborate:

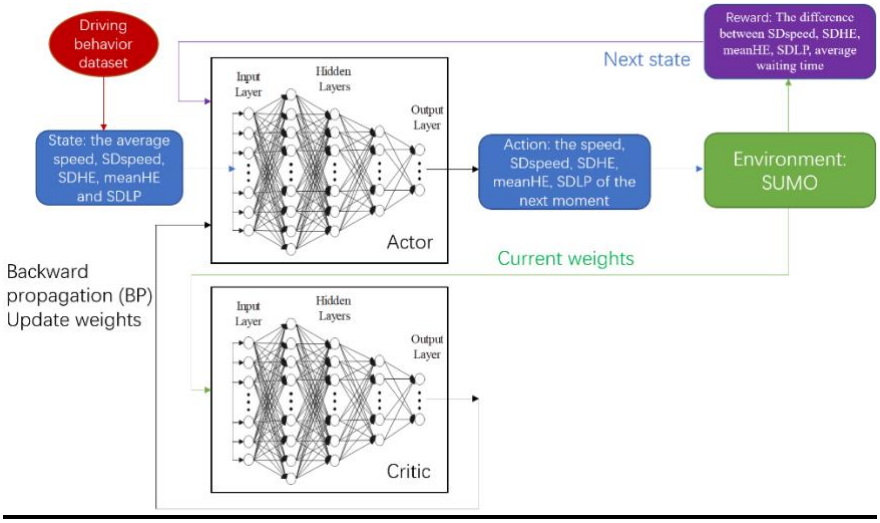
A2C – The Advantage Actor Critic algorithm is somewhat of a merge between the value-based and policy-gradient methods. Each iteration, it computes two values – the Critic function, which is the same function the DQN goes by, to minimize the error, and the Actor function, which takes the Critic's result in consideration and updates the policy the agent goes by.

PPO – The Proximal Policy Optimization algorithm does what on-policy algorithms do, which is to continuously update the policy, so that the agents using the policy will be guided to making the optimal decisions for the entire system, to maximize our reward OVERALL (opposed to maximizing it in the short term, like DQN)

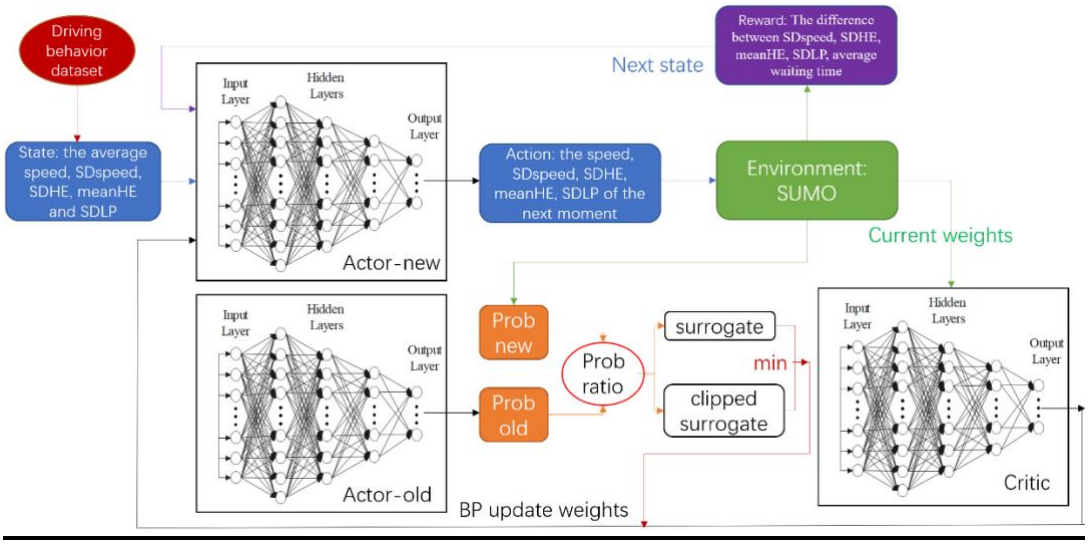
DQN structure:



A2C structure:



PPO structure:



State setting:

We used the average speed, SDspeed, SDHE, meanHE and SDLP in the automatic driving system as the state input.

Action setting:

We used the speed, SDspeed, SDHE, meanHE, SDLP of the next moment as the action space.

Reward setting:

The difference between SDspeed, SDHE, meanHE, SDLP, average waiting time at the next moment and SDspeed, SDHE, meanHE, SDLP, average waiting time at the previous moment. The reward values are the differences.

Experimental results – Single Agent (Intersection) problem:

We first run Random agent, which makes random actions, which aren't a result of his previous actions & rewards, and fixed agent, which gives each vehicle a fixed action such as "speed+=5", and compare their performance.

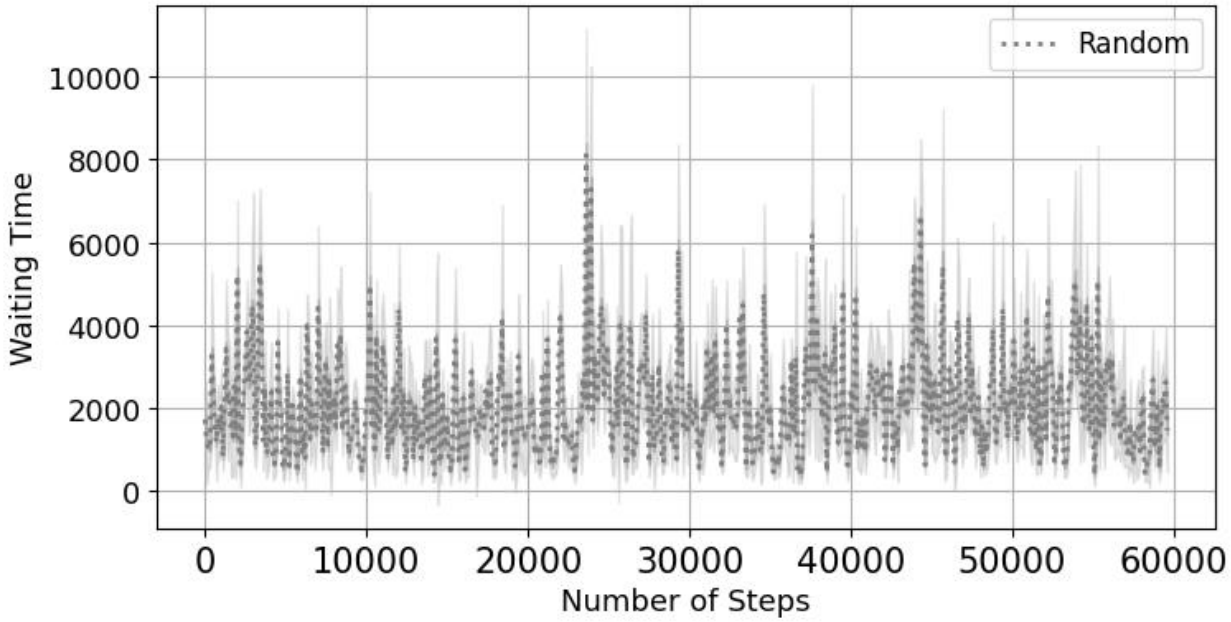


Figure 1. Random action space

Figure 1 clearly shows that random action space does not learn anything (as expected), and the waiting time doesn't improve overtime.

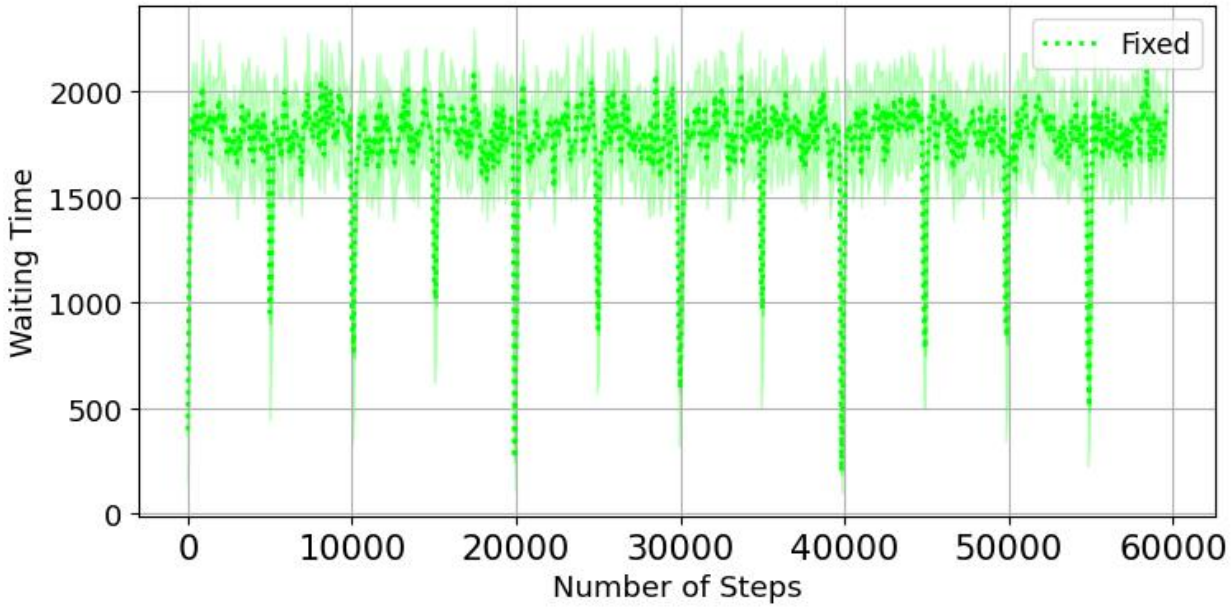


Figure 2. Fixed action space

As shown in the figure 2 and 3, fixed action space has much better performance than random action space in terms of speed (e.g., avg. waiting time), but still – as it's fixed, it doesn't improve overtime, and doesn't learn about the real flow of the traffic in the intersection.

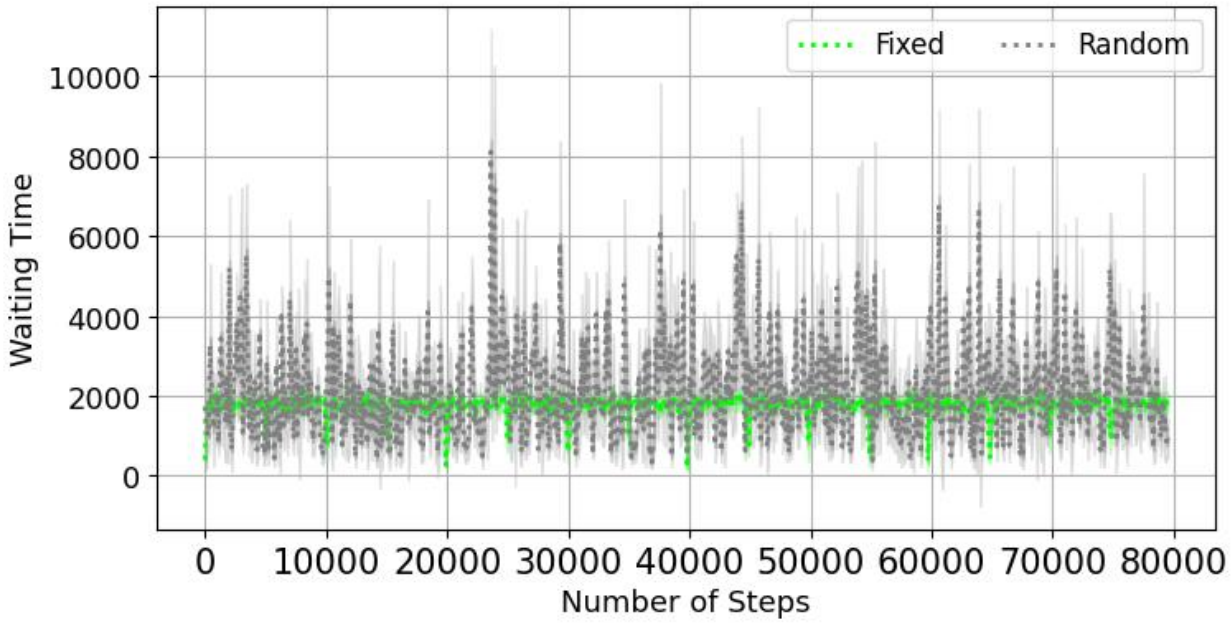


Figure 3. Comparison between random and fixed action space

Now, we move on to trying to manage the driver behaviors (SDspeed, SDHE, meanHE, SDLP) to be responsive of the actual traffic in the intersection, using DQN:

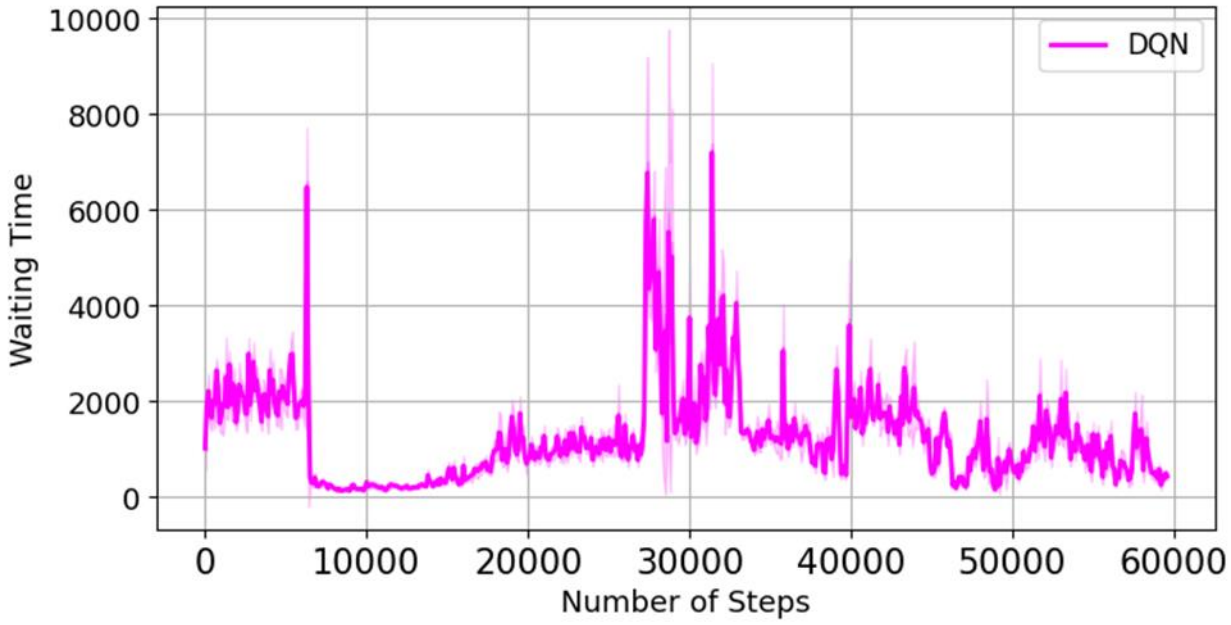


Figure 4. DQN result of single agent traffic

DQN seems to converge and improve its results overtime, but the final results aren't outstanding. The DQN algorithm is an off-policy algorithm, which tries to maximize the reward it gets in each iteration separately. This approach gets us pretty good results, but not optimal. Optimal results can be achieved with algorithms that learn a POLICY to go by (like a manual for the agents), that way we achieve a MACRO view instead of a MICRO view in the DQN algorithm.

After realizing on-policy algorithms might perform better in this problem, let's take a look on their result, in the next page. Let's check the A2C performance:

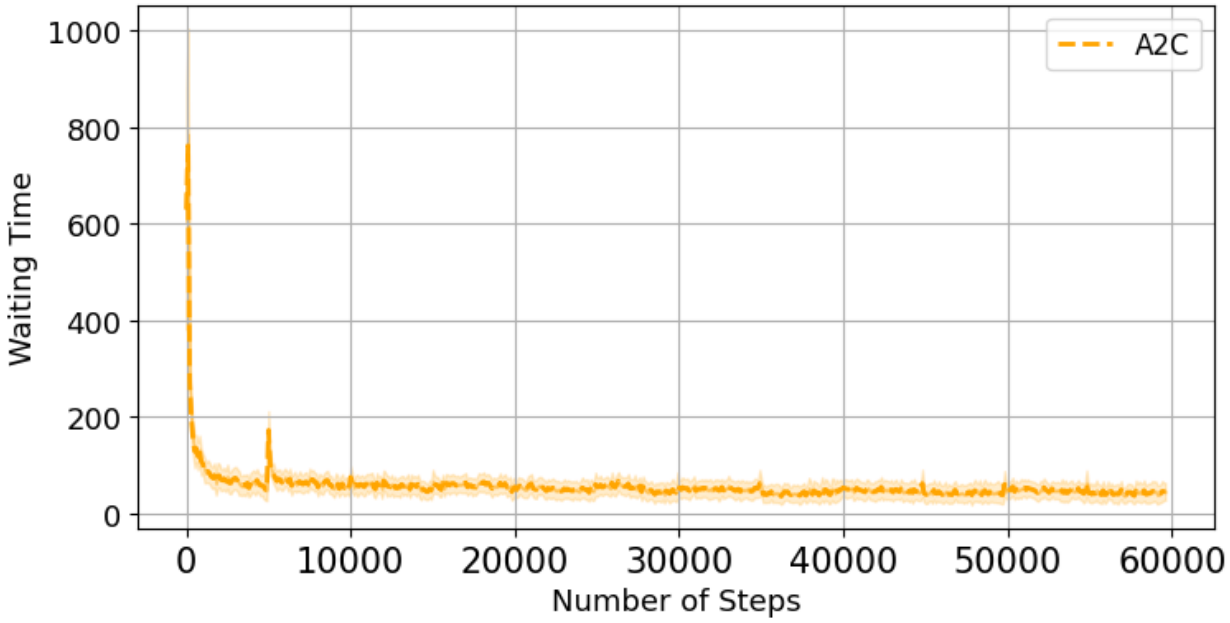


Figure 5. A2C result of single agent traffic

As we can see in the chart, A2C algorithm manages to successfully deal with the single intersection driving behavior modelling problem and controls each vehicle in a way that minimizes almost completely the cars' waiting time.

Let's also check PPO performance:

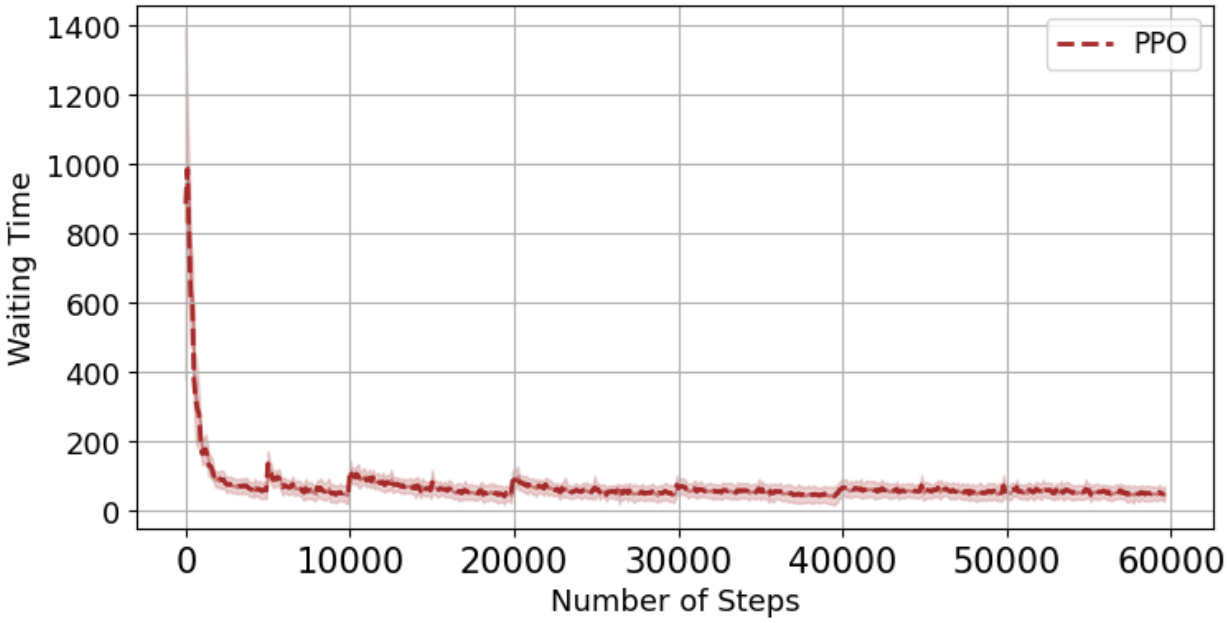


Figure 6. PPO result of single agent traffic

As they are relatively similar, PPO algorithm also performs well in the single agent problem.

Single agent conclusion:

As shown in Figure 7:
Random Action – cannot learn any driving behavior.

Fixed Action – cannot learn any driving behavior, the result don't improve overtime.

DQN – Converges successfully overtime and is better than fixed, but still performs worse than on-policy algorithms.

A2C, PPO – Waiting time decreases massively and end up minimized – solves and deals well with one intersection.

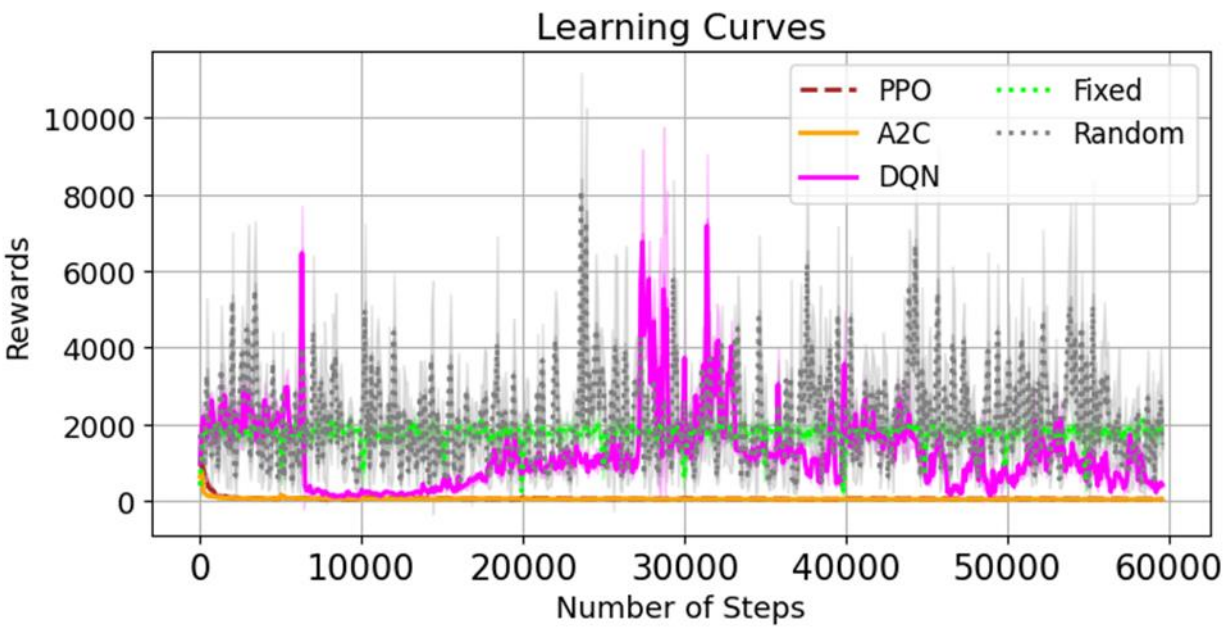
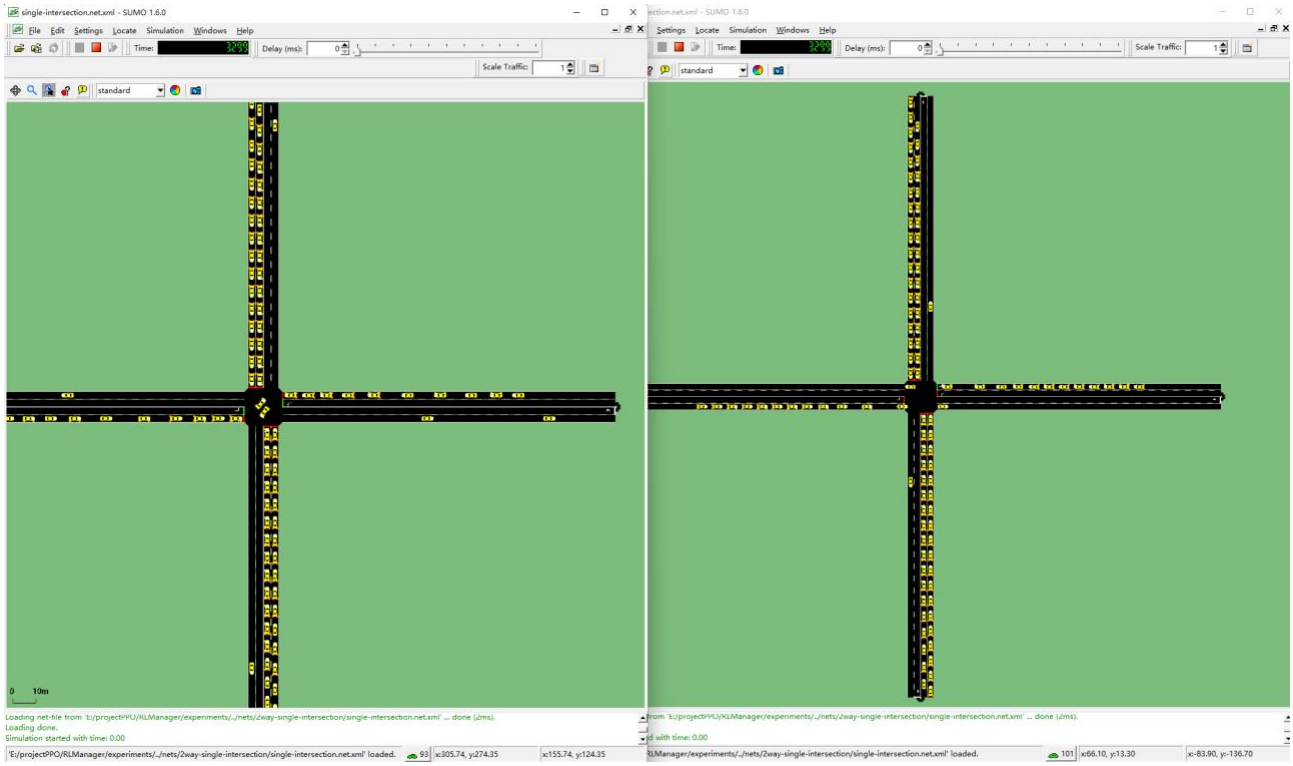


Figure 7. Reward comparison of different RL of the single agent

Multi-Agent:

After finding a good driving behavior model and minimizing waiting time in the Single Agent problem (just one intersection), we move on to the multi-agent problem – The real world. In real life, there are a lot of intersections, where one road you take in an intersection leads you to another in the next one.
We'll attempt to deal with this problem as well.



In the picture above, we can see two intersections (therefore we have two agents), where each agent is controlling its own intersection. Here are the results:

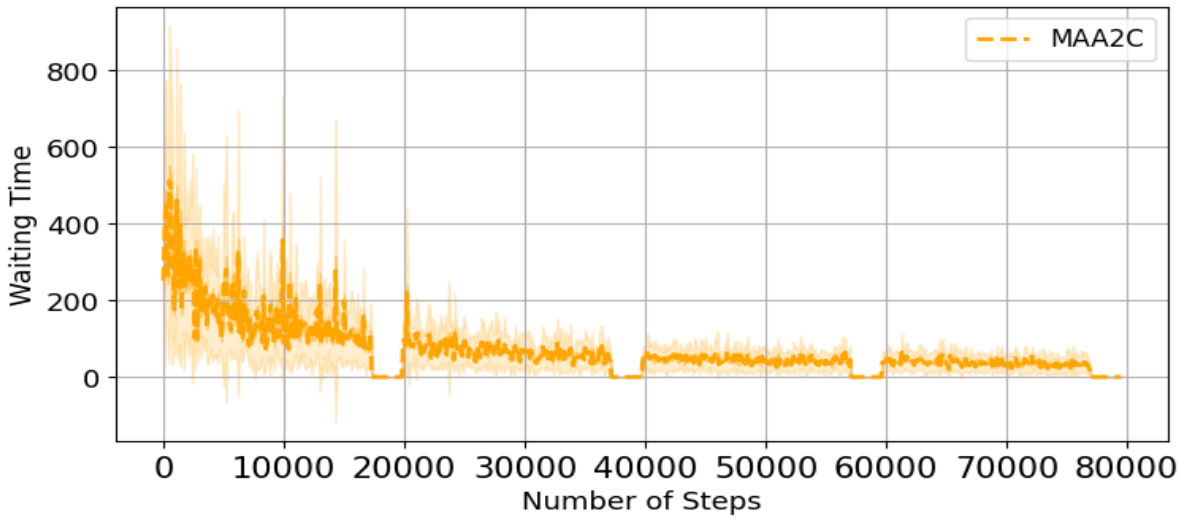


Figure 8. A2C result of the multiple agent

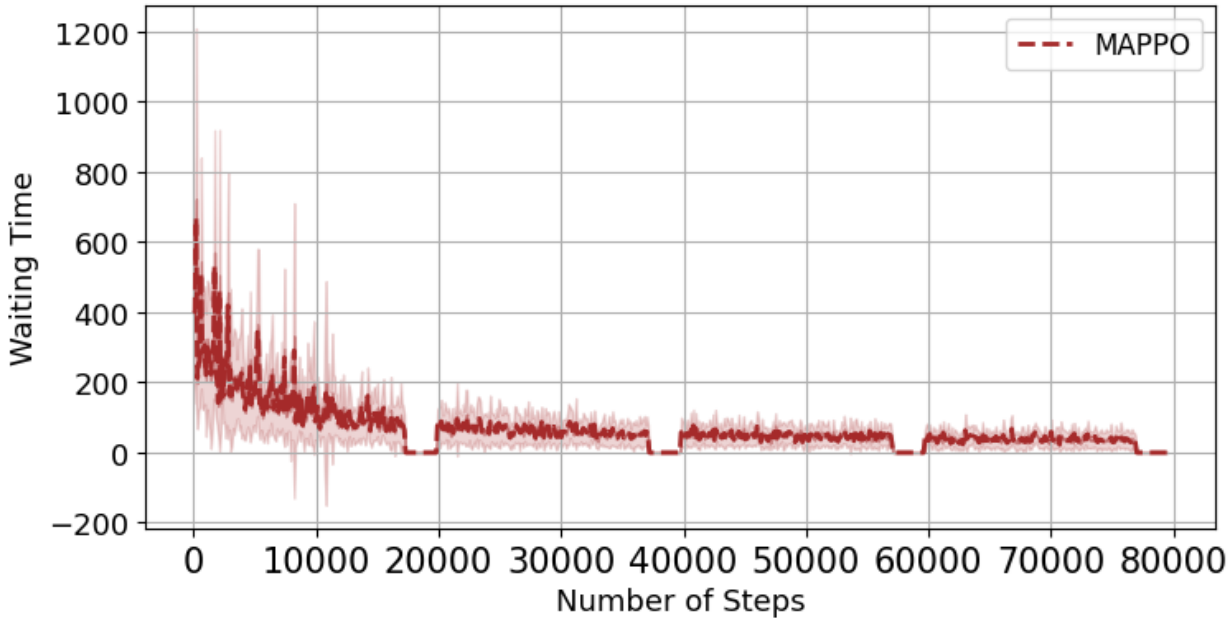


Figure 9. PPO result of the multiple agent

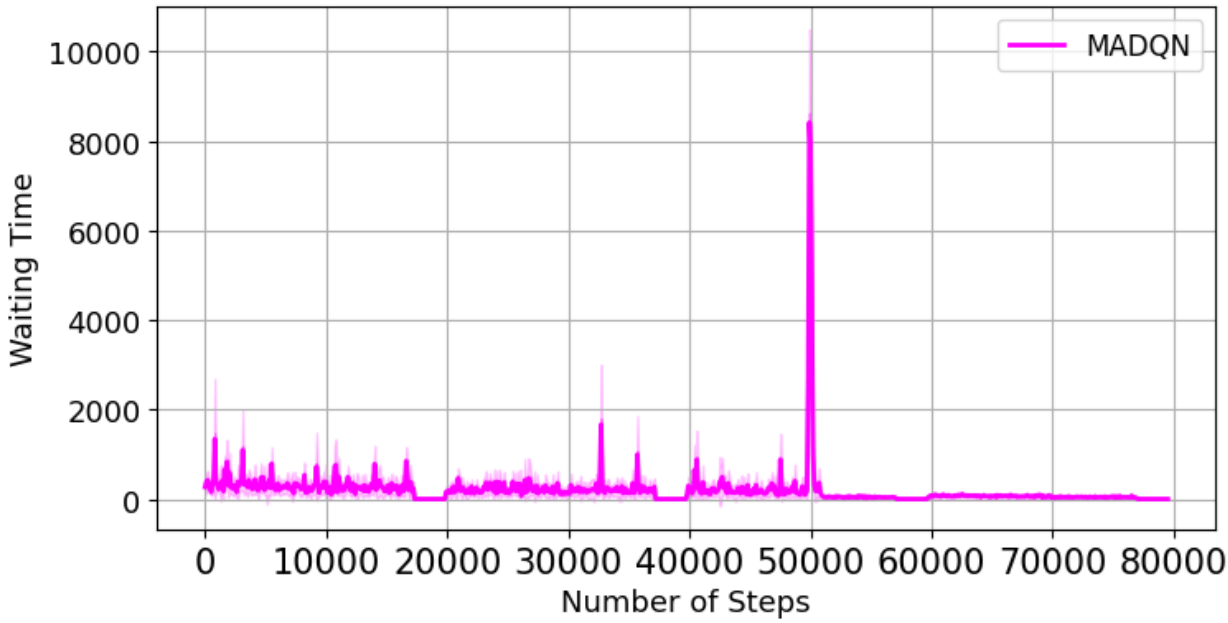


Figure 10. DQN result of the multiple agent

What can we see?

DQN again successfully converges after a decent amount of learning, but is again worse than A2C and PPO (look at the y axis' scale).

As for A2C and PPO, not only we succeed at solving this problem, the waiting time for all cars in all intersections combined is the same as it was in the single agent problem.

How is this possible?

The cause of this is the fact that in single agent, the agent has no idea where the traffic will come from. In Multi Agent we have a big advantage – the agents communicate, and tell each other – "Cars are coming from your __ side, be aware". This way the Multi Agent problem handles far more cars and intersections, but can maintain similarly low total waiting times. After seeing the great result with 1 & 2 intersections, we can look at the real world, where there's a net of intersections, all connected to each other, where cars flow in every direction.

Model Comparison Conclusion:

We've seen that PPO and A2C work better than DQN on this case of problem.
Why?

A – A2C tends to succeed in scenarios where information needs to be transferred between agents in an environment.

B – This problem fits the on-policy model better, since it's very behavior-based, and doesn't converge well when actions are always taken in a greedy manner (like DQN).

C – Having a policy that is learned and can guide the agents what to do next helps us here, since it provides a lot of information to the agents, who can now make their decisions based on the entire environment instead of just making sure they do the best thing for their own "small world" in the specific moment in time. Also, a policy to guide the agents provide us with a macro view that is capable of capturing the whole scene, instead of each agent optimizing itself just by the next move.

Conclusion – Having a policy that learns from a macro view overtime works far better in this problem than having agents who each optimize their own benefit in every single independent action.