Final Project Report for CS 175

Project Title: Reinforcement Learning for Optimal Traffic Signal Control
Project Number: 7
Student Names:
Yiqian Li, 001099880,yiqianl4@uci.edu
Tong Zhang, 001101743, tongz31@uci.edu

1. Introduction and Problem Statement

Urbanization has led to the exponential growth of vehicles on the road, resulting in increased at road intersections. Traditional traffic signal control systems, which often operate on fixed or pre-schedules, struggle to adapt to the fluid dynamics of traffic flow, and hence, cause inefficiencies such as increased waiting times, traffic jams, and higher emissions. With advancements in artificial intelligence, reinforcement learning techniques, we proposed an adaptive and dynamic solution to optimize traffic control systems at intersections. This project aims to leverage various reinforcement learning approaches in conjunction with the Simulation of Urban Mobility (SUMO) traffic simulator, a powerful tool for modeling and visualizing traffic scenarios. The focus will be on selecting the most appropriate traffic light phase to enhance traffic efficiency.
The primary challenge this project seeks to address is the optimization of traffic signal phases at road intersections, specifically, the selection of action space is among four distinct phases: North-South Advance, North-South Left Advance, East-West Advance and East-West Left Advance. The optimization criterion is defined as the aggregate waiting times of vehicles, which are randomly generated following straight, left, or right trajectories, on each lane. To solve the problem, the project trained a reinforcement learning agent to process graphic data generated from the SUMO traffic simulator and generate output reflecting traffic efficiency. The agent's action space, defined by the traffic signal, is influenced by multiple reinforcement learning methods, such as DQN, Double DQN, Dueling DQN and Proximal Policy Optimization (PPO), integrated with exploration strategies including greedy, epsilon-greedy, and Upper Confidence Bound (UCB). The effectiveness of these methodologies will be evaluated based on the total waiting time of generated vehicles within a fixed time period.

2. Related Work

Numerous methodologies have been employed in the past to tackle the problem of traffic signal optimization. In recent years, machine learning and particularly reinforcement learning algorithms have drawn attention for their ability to provide dynamic and adaptable solutions. For instance, as referenced later, Vidali implemented a deep Q-learning agent for traffic signal control. It made use of the Q-learning equation $Q(s,a) = reward + gamma \bullet max\ Q'(s',a')$ to update the action values and a deep neural network to learn the state-action function.
In the context of this body of work, our project seeks to build upon this existing algorithms, expanding the application from solely Q-Learning to include DQN, Double DQN, Dueling DQN, and PPO as well. Moreover, we aimed to systematically evaluate the performance of these methods.

3. Data Sets

The primary data source for this project will be generated using the Simulation of Urban Mobility (SUMO) software. This simulation tool allows us to create and manage complex traffic scenarios, providing us with control over various variables such as

vehicle generation, route selection, traffic light phase, among others.
In terms of training Dataset, it will be based on a built-in map file in SUMO, specifically the 'environment.net.xml'. This map will serve as the virtual environment where the reinforcement learning agent is trained. To mimic real-life traffic conditions and improve the model's generalization, vehicles will be generated according to a Weibull distribution. This distribution, often used in reliability analysis and life data analysis, has the flexibility to assume a variety of shapes and is suitable for modeling real-life phenomena such as vehicle arrival times at an intersection. The exact parameters of the Weibull distribution, such as scale and shape, can be adjusted to simulate different traffic conditions.
While in regard to testing Dataset, currently we are using the same map as the training dataset. However, in our future work, in order to evaluate the model's performance and its ability to generalize to different environments, we will use real-world maps, such as the Irvine map or maps from other cities. These maps will be transformed into a SUMO-compatible format using SUMO's NetEdit tool. The testing vehicle traffic will also be generated using different distribution settings, allowing us to evaluate the model's performance under various traffic conditions.

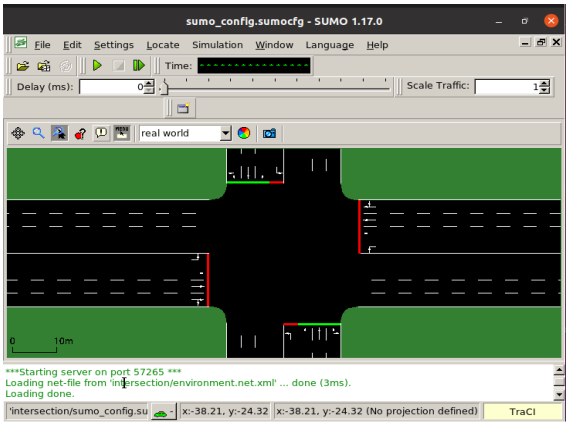

Figure 1 built-in map in SUMO

The use of SUMO as a platform for both training and testing allows us to have an extensive control over the traffic conditions and facilitates the evaluation of reinforcement learning strategies under a multitude of scenarios. This methodology can be particularly effective in identifying the best strategies for specific traffic conditions.

4. Description of Technical Approach

To simulate the traffic environment and train our model, we use the simulation software SUMO and the python package traci to connect it.
We referred to a project Deep-QLearning-Agent-for-Traffic-Signal-Control written by Andrea Vidali [6].
To simulate the interactive environment, we generate 1000 cars in each training episode. The appearance time of the cars follows the Weibull distribution. The cars will choose to move straight with 75% possibility and turn left or right with 25% possibility. They will also choose the roads randomly with equal possibility. Their behavior will be dumped into episode_routes.rou.xml for SUMO to simulate.
To retrieve the current state, we use traci.vehicle.getIDList to get the ids of all the cars. For each id, we use traci.vehicle.getLanePosition to get its position on the lane. We divide the length of the lane into 10 cells and translate the position into one of the cells. We use traci.vehicle.getLaneID to know which lane the car is in. Then, we combine the two messages and get a car position in the range of 0 to 79. The car that has not appeared will have a position less than 0 and be discarded. We create a bitmap with length of 80

and write 1 if there is a car with the corresponding car position.
We define our action space to be 4, which means the traffic light that will turn green. If the action is different from the previous one, meaning the traffic lights change, we let the previous traffic light turn yellow and later let the current traffic light turn green.
We define our reward as the total weighting time after 1000 cars passing through the crossroads. We use traci.vehicle.getAccumulatedWaitingTime to get waiting time for each car. We will also make sure the car is on the map and on the incoming roads. Otherwise, we will discard its waiting time.
To build a model that will output the proper policy according to the given state, we think of two directions. We can use a value-based method to train a value function and take an action based on value. Or we can directly train the policy. To get the experience, we use the epsilon-greedy method to choose actions, which can balance exploration and exploitation.

For the value-based method, considering our state space is 2^{80}, we decide to use deep Q-network to match the state with value, which is built through tensorflow.keras. We have to generate a large number of cars to get one reward. To efficiently use the data and also break down the correlation of the order of transitions, we use experience replay after simulation enough times to train our model. The DQN mode given by Vidali satisfies our goals. We firstly apply it to solve the problem. It will simulate the traffic lights choosing for enough times to allow all the generated cars to pass through the crossroads. All the experience will be stored in memory. Then, the experience will be retrieved. The states will be input into a neural network to get the values. The values will be updated according to the rewards and feedback to the neural network for training.

$$Q(S_t, A_t) = r_t + \gamma max_a Q(S_{t+1}, a) \quad \text{Eq.1}$$

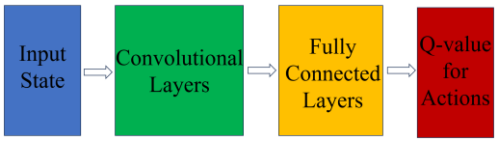

Figure 2 DQN Q-network structure

We notice that there are some limitations in DQN. During learning, action values try to chase a varying reward due to weights change, prone to instability. Therefore, we built and applied the Double DQN (DDQN). DDQN introduces a target Q-network with exactly the same structure as training Q-network but with different weights. When training the model, we only update the weights of training Q-network every time and will only copy the weights of training Q-network to target Q-network after certain episodes. As a result, the target value of the reward is relatively fixed during a period of time when the target network does not change, increasing the stability of learning.

$$Q(S_t, A_t) = r_t + \gamma max_a Q'(S_{t+1}, argmax_a Q(S_{t+1}, a)) \quad \text{Eq.2}$$

The model can also be modified to systematically distinguish which rewards are brought by state and which are brought by actions, leading to the Dueling DQN. The output of Dueling DQN is different. Dueling DQN algorithm includes two branches, which are the state value of the state and the advantage value of each action. The final output of Dueling DQN is the combination of the two branches.

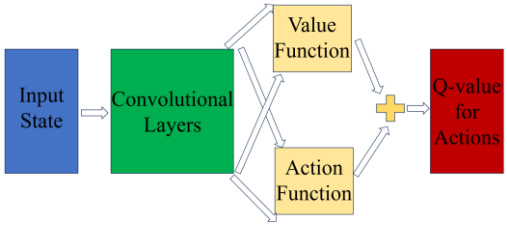

Figure 3 Dueling DQN Q-network structure

The policy-based method can avoid an intermediate step of learning a value function. To guarantee the stability of learning by avoiding policy updates that are too large, we choose to apply Proximal Policy Optimization (PPO) architecture.

$$J^{CLIP}(\theta) = E[min(\frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)})\hat{A}_{\theta old}(s,a), clip(\frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)}, 1-\epsilon, 1+\epsilon)\hat{A}_{\theta old}(s,a)] \text{ Eq3.}$$

We are interested in the performances of four different techniques. We compare their training converge performances, training rewards and testing rewards.

5. Software

(1) training_main.py (Written by Vidali)

Integrate all the modules needed for training, simulate for each episode and save training results.

(2) training_cont_main.py (Written by Li, referred to training_main.py from Vidali)

Continue to train a saved model. It was designed due to the computational limitation of colab, which allows the model to be continuously trained after disconnection.

(3) training_simulation.py

Simulate the interaction with the environment based on SUMO to train the model.

① function run (Originated from Vidali and modified by Li)

The main function is to simulate the interaction with the environment within one episode. It will choose an action according to current policy and state, calculate the rewards and store them into memory for replay, which will be repeated for self._max_steps times. The total reward will be stored as the training reward. Then, it will call function replay to train the model for self._training_epochs times. To make it fit the DDQN and Dueling DQN, Li added an update functionality for target Q-network after self._update_target_epochs episodes. Li modified it to compare the best history reward and output the best model instead of the last one.

② function _get_state (Written by Vidali)

Retrieve the state of the intersection from SUMO. It will create a bitmap called state with length of 80. Because there are four roads in the crossroads. Each road has two lanes for moving straight or turning right and turning left. The maximal number of cars in each lane is ten. If there is a car in the position, the index of that position will be set as 1, otherwise will be 0.

③ function _update_target_q_network (Written by Li)

Make the weights of target Q-network equal to training Q-network.

④ function _replay (DQN part was written by Vidali. DDQN, Dueling DQN and PPO parts were written by Li)

Retrieve a group of samples from the memory and train the model.

For DQN, it will directly update Q-network based on training Q-network.

For DDQN and Dueling DQN, it will choose the best action according to the target Q-network and use it to train the training Q-network.

For PPO, it will choose the best action within the proper changing ratio compared with old action.

The result will be feeded back to the CNN network to train the model.

(4) model.py

To build the whole reinforcement learning model for four types of methods.

① function _build_model

Build the neural network model. There are three types of models. The first one with input size 80 and output size 4 written by Vidali is a single CNN network. It can match states and actions for DQN, DDQN and policy networks for PPO. The second one with input size 80 and output size 4 written by Li has hidden layers for state value and action value and combines them for output, which is designed for Dueling DQN. The third one with input size 80 and output size 1 written by Li is the value network for PPO. Its initial weights come from a well-trained DDQN network to speed up the training. The structure of the three models are shown in the following figure.

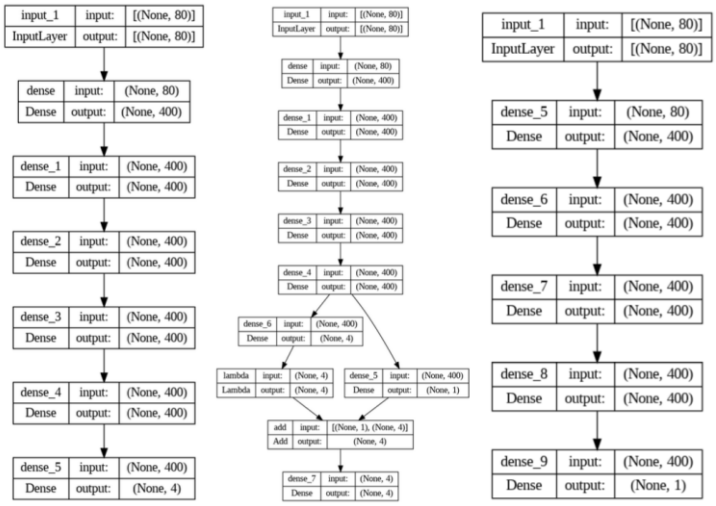

Figure 4 Three types of nerual networks. From left to right, the first is Q-network for DQN, DDQN, policy networks for PPO; the second is Q-network for Dueling DQN; the third is value network for PPO

② function _load_my_model (Originated from Vidali and modified by Li)

Load the existing model for continued training or testing.

③ functions predict_one and predict_one_value

DQN part was written by Vidali. DDQN, Dueling DQN and PPO parts were written by Li.

Given the state and output a list of actions value.

④ functions train_batch, train_policy_batch, and train_value_batch

DQN part was written by Vidali. DDQN, Dueling DQN and PPO parts were written by Li.

Given the state and updated action, fit the model.

(5) memory.py (Written by Vidali)

Store the simulation results for replay to retrieve.

(6) generator.py (Written by Vidali)

Randomly generate cars according to weibull distribution in the map. Store the simulation process in episode_routes.rou.xml to visualize in SUMO.

(7) testing_main.py, testing_simulation.py (Written by Vidali)

For testing. Similar to training_main.py and training_simulation.py. But will not train the model and will only simulate once.

6. Experiments and Evaluation

(1)Training

For the training part, we set total episodes as 40, training episodes as 200, the number

of cars generated per training episode as 10000, the maximal signal changing times as 5400, learning rate as 0.002, gamma for reward as 0.75 and target Q-network updated episodes as 10.

The setting of hyper-parameters is decided after several attempts. Dueling DQN did not converge under learning rates of 0.001 and 0.0001. When training episodes equal to 100, the training rewards vibrate largely while equal to 500 will cost lots of time with trivial training upgrades. DQN, DDQN, Dueling DQN suffer from exponentially increasing training time, which limits the number of total episodes.

The final training results are shown in the following four figures. The DQN resulted in the final reward of -5391 and best reward of -4748 in the 39th episode.The DDQN resulted in the final reward of -4241 and best reward of -4032 in the 38th episode. The Dueling DQN resulted in the final reward of -5963 and best reward of -5823. All of the three models showed a trend to improve during training. However, the PPO did not converge and could not be further trained after 10 episodes due to extremely long training time. Therefore, we had to discard it and not test it.

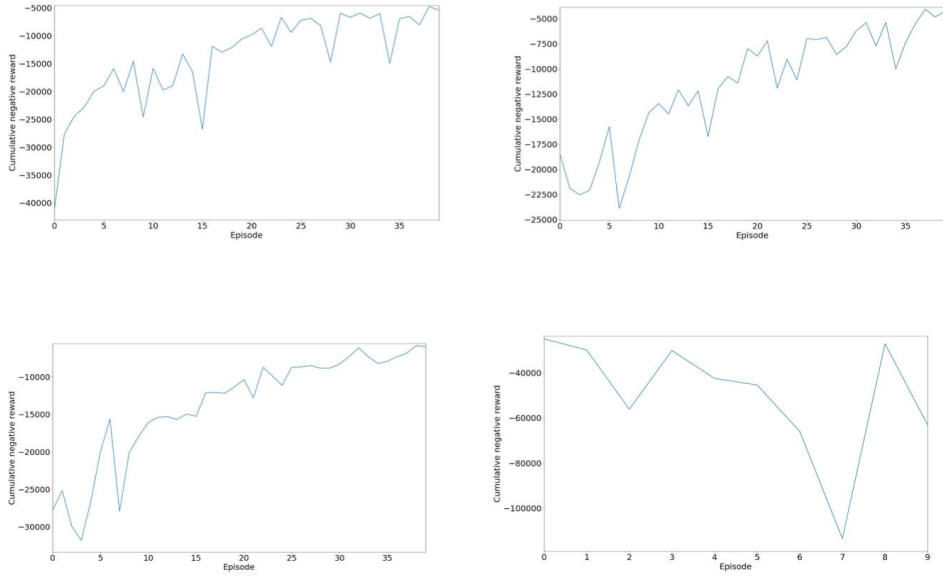

Figure 5 The reward figures during training. From left to right and from top to down, the first is from DQN; the second is from DDQN; the third is from Dueling DQN; the forth is from PPO.

(2) Testing

In terms of testing part, we test each model on testing datasets, which follow the same configuration of the training dataset. Based on the definition of rewards, we added up all the waiting time of generated vehicles from each lane and compare their values to evaluate different models performance on testing sets. The testing results are as follows:

--DQN with an aggregated rewards: -11,612

--Double DQN with an aggregated rewards: -3,819

--Dueling DQN with an aggregated rewards: - 4,336

The testing results showed that Double DQN presented the best performance as the aggregated waiting time of generated vehicles is the smallest, while Dueling DQN showed a slice worse performance compared to Double DQN. However, when considering the baseline method DQN, both Double DQN and Dueling DQN improved the performance significantly.

7. Discussion and Conclusion

We find that Double DQN and Dueling DQN can solve the traffic signal control problem efficiently. With target Q-network, Double DQN can perform even better in

training and testing. However, the baseline DQN cannot solve the problem in an efficient way. And the PPO cannot converge even within such a small action space. Under our approach, the reward is sparse. However, it is also difficult to decide when we should collect the total waiting time of all the passing cars.

There are some directions that can be explored in future. First of all, our map is a single crossroads. More testing scenarios, such as real-life maps should be implemented to evaluate the models' performance and generalize to other environments. Different types of map can be tried with reinforcement learning techniques. In addition, the neural network inside the model is a simple CNN network. The effects of different neural networks are worth comparison to achieve a better performance. Lastly, future work can also involve the development of a robust evaluation framework that considers other performance metrics beyond waiting times, such as fuel consumption, emission levels, and pedestrian priority and safety. These metrics would provide a more comprehensive understanding of the system's impact, potentially leading to a traffic signal control solution that balance efficiency with environmental and safety considerations.

8. References

[1] Vidali.A. https://github.com/AndreaVidali/Deep-QLearning-Agent-for-Traffic-Signal-Control