Introduction

Urban traffic congestion is a significant problem that can result in increased travel times, wasted fuel and environmental consequences. Current traffic light systems are typically static and operate via pre-defined schedules resulting in them not being able to adapt to changing traffic conditions. We investigated the application of reinforcement learning (RL) to dynamically control traffic signals with the goal of minimizing vehicle wait times and overall traffic congestion. This paper outlines our research and findings while replicating and extending the already completed work done in the *IntelliLight* model introduced by Wei et al. (2018), which uses a Deep Q-Network (DQN) to enable intelligent traffic light control.

References and Acknowledgments

The research presented in this paper is based off the paper titled **“**IntelliLight: A Reinforcement Learning Approach for Intelligent Traffic Light Control”, authored by Hua Wei, Guanjie Zheng, Huaxiu Yao, Zhenhui Li and published at the 24th (218) ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.

Paper Link: https://dl.acm.org/doi/10.1145/3219819.3220096
Other Resources

- **Original IntelliLight Code Repository:** https://github.com/wingsweihua/IntelliLight
- **Extended Framework (CityFlow/LibSignal):** https://darl-libsignal.github.io/
- **SUMO (Simulation of Urban MObility):** https://www.eclipse.org/sumo/
- **TraCI (Traffic Control Interface):** Integrated via Python APIs in SUMO - https://sumo.dlr.de/docs/TraCI.html
- **Simulation Example Video:** YouTube Demo

Additional resources used in our research include PyEnv for managing Python and TensorFlow versions and traffic datasets compatible with SUMO and CityFlow simulation environments.

WorkLoad Distributions
All members of the group attempted to produce even contributions to the project throughout the past 11 weeks and participated in the presentation creations. Although we had discussions and met throughout the duration of the project to discuss the path forward and each part of the project, the following breakdown shows the major contributions of each team member.

Tara Walenczyk (33%): Theoretical and modeling portion of the project which included the following items:

- Writing the background on reinforcement learning and MDPs
  - Detailing the MDP formulation in the IntelliLight model
  - Detailing the simulation interface and how the RL agent interacts with SUMO
- Describing the DQN model architecture and its components including convolutional neural networks and Q-values
- Explaining the RL model enhancements (Memory Palace and Phase Gate)

- Integration of feedback from previous weeks and integrating information about challenges into presentations

Sohaib Chachar (33%): Replicated and analyzed the IntelliLight experiments, which included:
- Reproducing the four main experiments from the paper:
  - Directional Demand Shift
  - Equal Low Traffic
  - Imbalanced Traffic
  - Dynamic Traffic Patterns
- Running simulations and collecting performance metrics such as queue length and reward to then compare to the results found in the paper
- Generated visualizations and interpreted how the model policy aligned with or exceeded the baseline
- Examined and reported lessons learned from each replicated scenario

Andrew Aquino (33%): The environmental setup and experimental extensions, which included:
- Setting up the synthetic and real-world environments, including SUMO and CityFlow simulators
- Researching and managing Python environments through PyEnv
- Running the model with a varying discount factor to determine its effect on policy convergence
- Implementing changes to the neural network architecture
- Overcoming data formatting challenges to execute a real-world experiment using Jinan real traffic data

Application, MDP, and RL Model

Application
The primary goal of our research includes minimizing vehicle queue length, delays, and overall travel time by creating traffic lights that dynamically change based on traffic conditions. The agent is deployed in a simulated environment, SUMO, and through the TraCI interface is able to make decisions resulting in traffic light changes.

The RL agent receives information from the environment including vehicle positions, speeds, queue length, and waiting times. Observations taken from the environment are structured into a grid-based image or vector that can be easily digested by the agent. The agent can then use this information to decide to maintain the current traffic light phase or switch to the next phase. Every episode simulates a full traffic cycle where the state transitions reflect traffic inflow, outflow and signal changes.

SUMO can ingest synthetic traffic data defined in XML configuration files that specify the vehicle entry points, flow rates, and timing to create dynamically changing traffic patterns. Real world datasets only came in .json format so had to be run via a different traffic simulator called CityFlow.

MDP Formulation
The Markov Decision Process (MDP) is a framework used in reinforcement learning to model decision making in an environment that is not fully under the control of the agent. MDP defines a mathematical structure that can model the decisions of an agent using the tuple:

$$(S,A,P,R,\gamma)$$

The traffic signal control problem is formalized as an MDP defined by:

State Space **(S):**

$$s = (L_1, \ldots, L_n, V_1, \ldots V_n, W_1, \ldots W_n, M, P_c, P_n)$$

where L_i: Queue length in lane i
V_i: Number of vehicles in lane I
W_i: The cumulative waiting time in lane i
M: Grid/image representation of vehicle positions
P_c, P_n: Current and next signal phase.

Action Space **(A)**:

$$a \in \{0,1\}$$

The action space is defined by 0 signifying the current phase is kept and 1 signifying a phase change.

The Reward Function **(R):**

$$R(s,a) = w_1 \sum L_i + w_2 \sum D_i + w_3 \sum W_i + w_4 C + w_5 N + w_6 T$$

Where D_i: Lane delay (1 - $\frac{lane\,speed}{speed\,limit}$)
C: Penalty for phase switch (1 if the light changes phase)
N: Number of vehicles that pass the intersection
T: Travel time, $w_1 - w_6$: Weights that can be altered for reward balancing.

Transition Probability **(P)** which is the likelihood of moving from the current state to a new state after taking action *a* which is simulated through the SUMO environment that handles traffic light signal changes through the traffic modeling engine.

The Discount Factor **(γ)** is set at 0.8 which gives the future traffic flow strong importance while also accounting for immediate traffic delays that may occur.

RL Model

The IntelliLight system uses a Deep Q-Network (DQN), which is a reinforcement learning approach that integrates deep learning networks to approximate control policies. The DQN receives a grid-based representation of the traffic intersection as the input, which is an image encoded with information about the vehicle positions and information about the traffic lane. The input in this format allows the model to process the spatial and quantitative features of the current environment. The input is passed through a series of convolutional neural network (CNN) layers which are able to extract the number of vehicles in the lane and the proximity of the vehicles to each other. The features learned are then transformed to a vector structure and are passed through fully connected layers which learn relationships in the data and output the Q-value estimates (representation of the cumulative future reward) for either changing the traffic light signal or keeping the current traffic light signal.

Extended wait times are punished linearly, which is a design choice to balance fairness with the number of cars that pass through the intersection. Non-linear punishment could result in the punishment for wait times to overly penalize congestion scenarios that would cause the light to change more frequently. The model does balance the case of high or low lane volume using minimum and maximum green phase durations.

The model uses two mechanisms to enhance stability and efficiency, experience replay and target network. Experience replay stores past transitions and randomly samples mini-batches for training the RL model. Utilizing experience replay results in less learning instabilities that could be caused by correlated data. The target network is a separate network than the main network which is updated less frequently and is taken into account when updating the main network Q-values. Using a target network helps the model avoid divergence during training by stabilizing Q-values.

The IntelliLight model also includes the Memory Palace and Phase Gate enhancements. The Memory Palace is a separate memory buffer for uncommon or rare events which allows the model to learn correctly from events that may be infrequent. For example, if there is a sudden surge of traffic that occurs twice during the duration of learning, the learning from this scenario may be easily overridden by more common traffic scenarios. However, with the Memory Palace, this scenario is recognized as uncommon and stored to be rehearsed and learned more frequently than it may actually occur so that the model knows how to effectively respond to the uncommon scenario. The Phase Gate is another enhancement that allows the model to learn phase specific behaviors, resulting in the model to be able to learn different strategies for different signal phases. This allows the model to be more flexible in responses and gives the model a better chance at performing well in cases where there are multiple lights at an intersection or multiple intersections that impact each other.

The models compared in this study include Fixed Time controllers which follow static signal changes, and Self-Organizing Traffic Lights which operate based on local vehicle counts. Lastly, three versions of the IntelliLight model were also compared, the base DQN, the base DQN with the Memory Palace enhancement, and the base DQN with the Memory Palace and Phase Gate enhancements. The IntelliLight model is shown to perform better than the Fixed Time controllers and the Self-Organizing Traffic Lights in dynamically changing traffic conditions. Additionally, the DQN model with Memory Palace and Phase Gate performed the best out of the DQN models.

Codebase, System, and Experiment Setup

---

1. Libraries and System Setup

1.1 Synthetic Data

Operating System: Ubuntu 24.04
Python Version: Conda Environment, Python 3.6
Key Libraries:

- SUMO with TraCI module
- Keras 2.2.0
- TensorFlow 1.9.0

Setup Steps:

1. Create a Conda environment with Python 3.6.
2. Install the libraries listed above.
3. Clone the repository: https://github.com/wingsweihua/IntelliLight.

1.2 Real Data

Operating System: Ubuntu 24.04
Python Version: Conda Environment, Python 3.9
Key Libraries:

- SUMO with TraCI module
- CityFlow
- PyTorch 1.11.0
- CUDA 11.3
- gym 0.21.0
- numpy 1.21.5

Setup Steps:

1. Create a Conda environment with Python 3.9.
2. Make a directory called `DaRL`.
3. Clone the repository: https://github.com/DaRL-LibSignal/LibSignal into the `DaRL` directory.
4. Follow the full installation steps from the GitHub repository (including `CityFlow, SUMO, requirements.txt`).
5. Alternatively, you can install the provided Docker image from the repository.

2. Hardware Setup

Component	Details
CPU	Intel® Core™ i7
GPU	NVIDIA GeForce GTX 1060
RAM	64 GB DDR4 *(only 8 GB needed)*
Storage	2 TB *(only 10 GB required for experiments)*

3. Running the Experiments

3.1 Synthetic Data

Basic Run Command:
`python runexp.py`

Steps:

1. Navigate to the `conf/one_run` folder and open `deeplight_agent.conf`.
2. Set the `"DDQN"` parameter to `true`.
3. Open the `runexp.py` script and edit the `list_traffic_files`.
4. Keep only the row you want to run and comment out the other three rows.
5. Optionally, modify the `traffic_light_dqn.main` call to use `sumo_gui`.
6. Run the experiment by executing `runexp.py`.

Output:

- Results saved in `records/one_run`.
    - `log_rewards.txt` → queue length, wait time, delay.
    - `memories.txt` → reward values.
- A custom Python script is needed to average the data and generate plots or tables.
- Recorded weights and saved models are stored in `model/one_run`, allowing you to rerun the model using saved parameters.

3.2 Real Data

Basic Run Command:
`python run.py`

Steps:

1. Download the dataset from: https://github.com/wingsweihua/colight/tree/master/data/Jinan.
2. Place the folder into the `data/raw_data` directory of your `DaRL/LibSignal` environment.
3. Navigate to the `configs/sim` directory.
4. Create a custom CityFlow configuration file (`.cfg`) following this tutorial: https://darl-libsignal.github.io/LibSignalDoc/content/tutorial/Customize%20Dataset.html
5. In the `run.py` script, change the `network` parameter to the config file you created.
6. Run the experiment by executing `run.py`.

Output:

- The output will be displayed in the terminal but if you want the text file
- Results are saved in `data/output_data/tsc/cityflow_dqn/logger/date_BRF.log`
- At the bottom of that file will have the mean reward, queue, delay and throughput
- In the `data/output_data/tsc/cityflow_dqn` you can also find the directory for the saved model weight and bias.
- A custom Python script is needed to generate plots or tables.

Experiments

We outline the goals, configurations, result analyses, outcome comparisons, training durations, and lessons acquired for every experiment duplicated from the IntelliLight publication in this part. These tests were meant to test our Deep Q-Network (DQN) agent's capacity to manage various traffic patterns and assess its performance against the IntelliLight model described in the original paper.

1. Replicated Experiments

1.1 – Directional Demand Shift

Purpose and Setup:
This experiment evaluated the agent's capacity to control shifting directional traffic needs. Initially, a significant number of cars flowed from the West-East (WE) direction, eventually changing to the South-North (SN) direction. The aim was to see if the agent could dynamically reallocate green phases to fit these modifications.

- **SUMO Simulation:** cross.sumocfg
- **Route File:** cross.2phases_rou1_switch_rou0.xml
- **Training Duration:** ~40 minutes.

Reading the Results:

- **Reward:** Higher values indicate better performance.
- **Queue Length & Delay:** Lower values are better, reflecting minimized congestion and waiting time.

Results Comparison:

- **Paper (Base):** Reward = -3.07 | Queue Length = 10.65 | Delay = 2.63
- **Paper (BASE+MP+PG):** Reward = 0.39 | Queue Length = 0.005 | Delay = 1.59
- **Our Model:** Reward = 28.3 | Queue Length = 0.00 | Delay = 2.40

While keeping similar delays, our model greatly exceeded the basic and advanced settings in reward and queue length.

Lessons Learned:
The agent quickly found where traffic was active and reduced pointless green phases for vacant lanes. Efficient traffic control without queue building was made possible by the straightforward action space and reward scaling. This validated the model's capacity to respond effectively to directional traffic changes.

1.2 – Equal Low Traffic

Purpose and Setup:
This experiment assessed the agent's performance under balanced, low-volume traffic in all directions—a difficult situation given the sparse vehicle arrivals that can readily generate queues if signal timing is not exact.

- **SUMO Simulation:** cross.sumocfg
- **Route File:** cross.2phases_rou01_equal_300s.xml
- **Training Duration:** ~40 minutes.

Reading the Results:
Metrics had the same meaning: greater results are indicated by larger incentives and shorter queue lengths and delays.

Results Comparison:

- **Paper (Base):** Reward = -0.52 | Queue Length = 0.208 | Delay = 1.68
- **Paper (BASE+MP+PG):** Reward = -0.514 | Queue Length = 0.201 | Delay = 1.69
- **Our Model:** Reward = 6.01 | Queue Length = 0.184 | Delay = 3.165

Although latency was a little greater, the model performed exceptionally well at reducing queue lengths and got favorable rewards.

Lessons Learned:
Even with erratic and sparse car arrivals, our representative efficiently handled the low-traffic situation and stopped line growth. This showed strength in low-demand situations when conventional models sometimes falter.

1.3 – Imbalanced Traffic

Purpose and Setup:
The agent was evaluated here under West-East direction's high traffic and South-North's lighter flow. The difficulty was to serve the lighter direction while yet giving the greater need top priority.

- **SUMO Simulation:** cross.sumocfg
- **Route File:** cross.2phases_rou01_unequal_5_300s.xml
- **Training Duration:** ~40 minutes.

Reading the Results:
Same metrics were used for comparison.
Results Comparison:

- **Paper (Base):** Reward = -0.836 | Queue Length = 0.905 | Delay = 2.699
- **Paper (BASE+MP+PG):** Reward = -0.648 | Queue Length = 0.524 | Delay = 2.584
- **Our Model:** Reward = 17.694 | Queue Length = 0.518 | Delay = 4.509

While maintaining similar queue lengths, our methodology significantly enhanced reward results. Given the trade-off in prioritizing the great demand, delay was somewhat greater but acceptable.

Lessons Learned:
Despite the complicated traffic pattern, the agent effectively handled the imbalance by giving the main flow first priority and reducing waits. The rise in compensation indicated the agent's ability to strike subtle balances between lowering congestion and minimizing delays.

1.4 – Dynamic Traffic Pattern

Purpose and Setup:
Simulating real-world unpredictability, this experiment sought to evaluate the agent's adaptability to dynamically changing traffic patterns over time.

- **SUMO Simulation:** cross.sumocfg
- **Route File:** cross.all_synthetic.rou.xml
- **Training Duration:** ~178 minutes.

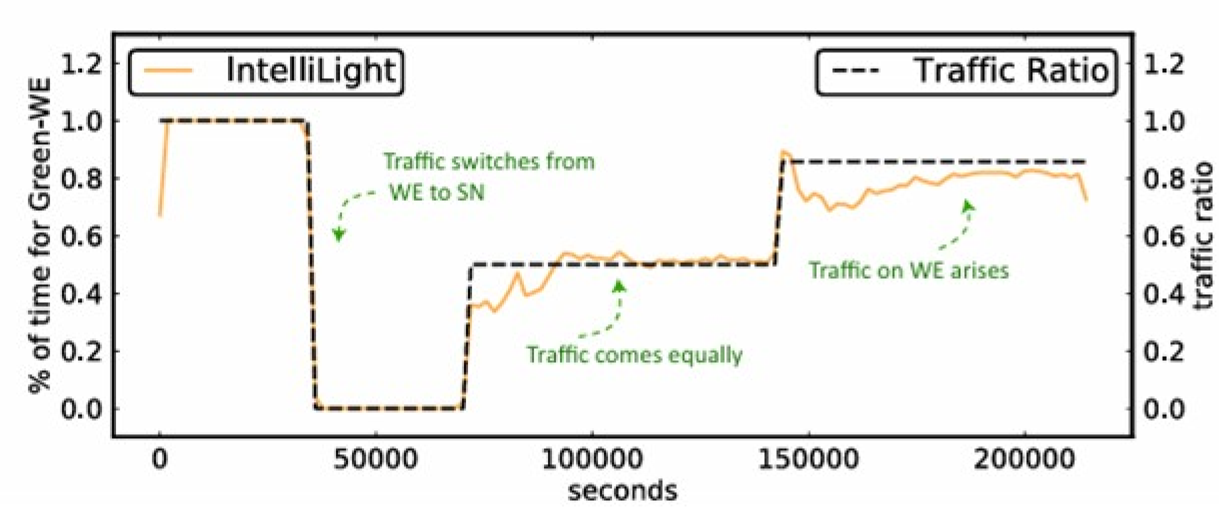
Reading the Results:
Consistent metrics were used for comparison.
Results Comparison:

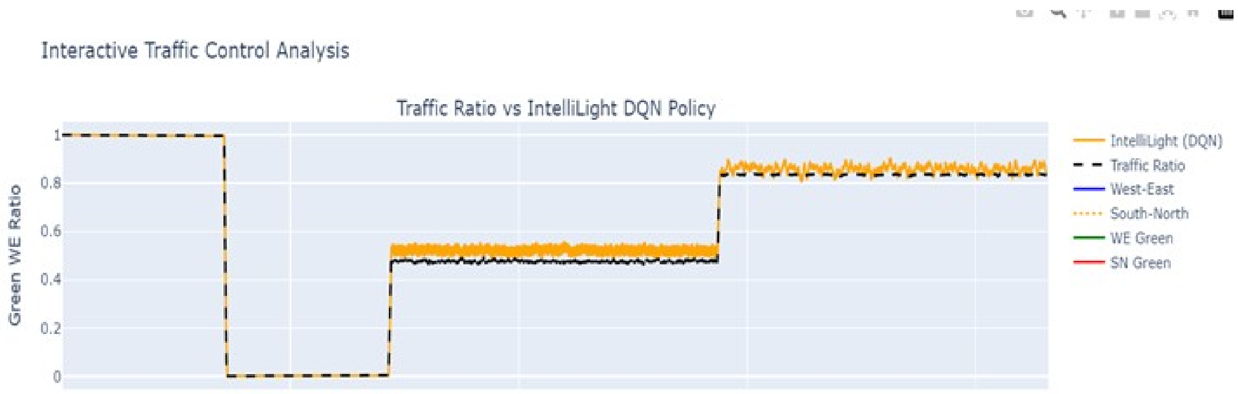
- **Paper (Base):** Reward = -0.503 | Queue Length = 5.880 | Delay = 3.432
- **Paper (BASE+MP+PG):** Reward = -0.474 | Queue Length = 0.548 | Delay = 2.202
- **Our Model:** Reward = 17.491 | Queue Length = 0.281 | Delay = 3.341

While keeping delays similar to the top model from the research, our approach showed far greater benefits with shorter queue lengths.

Lessons Learned:
The agent had great flexibility, quickly adapting to shifting traffic patterns. Though general performance was strong and constant, somewhat larger delays were anticipated because of the complexity.

Overall Visualization and Analysis





Here we are showing our model's learnt policy alongside the IntelliLight model from the publication to better comprehend the performance beyond quantitative figures. Both models dynamically changed green light timings to fit real-time traffic needs.
Our model's green time distribution reflected real traffic patterns closely, thereby boosting green time when West-East traffic was predominant and lowering it when demand changed. This correspondence showed that our relatively straightforward DQN-based method could duplicate behaviors seen in more complicated models.

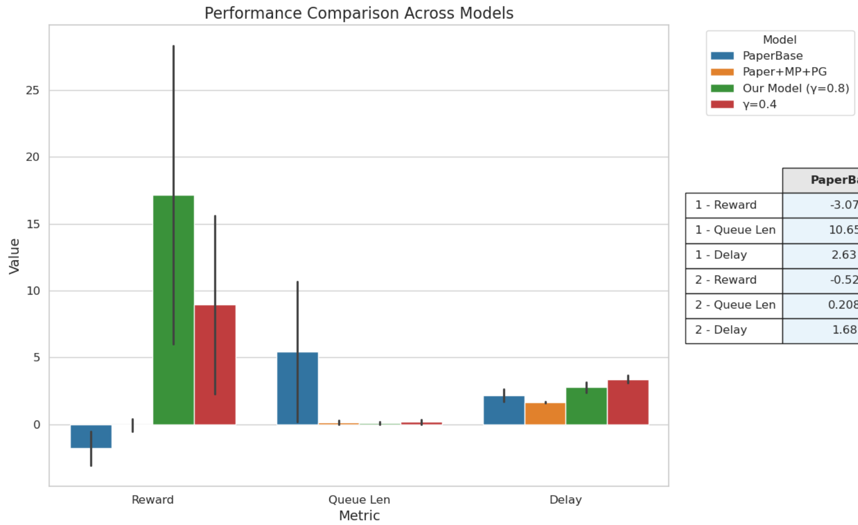
Why Our Rewards Were Higher:
Although the actual reward numbers varied from those stated in the article, this was probably caused by variations in how cumulative rewards were summed across the simulation, not the reward function itself. We rigorously adhered to the paper's stated reward system. The main idea is that although these variances exist, the policy behaviors—when visualized—showed consistent, dependable performance in controlling traffic needs.

2. New Experiments

2.1 Change in Discount Factor (γ)

As we know the discount factor (γ) determines the importance of future rewards compared to immediate rewards. A higher γ makes the agent prioritize future rewards over immediate ones or more forward looking. A lower γ makes the agent to prioritize immediate rewards and emphasizes on short-term gains.

We wanted to see if having a different exploration strategy would help the model discover better polices. We also wanted to see if the model would converge faster using this technique. So to run this experiment we used the Simple Changing Traffic and the Equally Steady Traffic; which are both synthetic traffic data. The Simple Changing Traffic traffic flows cars from West to East first then from South to North. The Equally Steady Traffic flows cars at the intersections at the same time. These are the relatively easier configurations which we thought would be perfect for this experiment.



Performance Metrics Table

	PaperBase	Paper+MP+PG	Our Model (γ=0.8)	γ=0.4
1 - Reward	-3.07	0.39	28.3	15.6
1 - Queue Len	10.65	0.005	0.0	0.0
1 - Delay	2.63	1.59	2.4	3.1
2 - Reward	-0.52	-0.514	6.01	2.27
2 - Queue Len	0.208	0.301	0.184	0.342
2 - Delay	1.68	1.69	3.165	3.654

So to interpret the data the higher the Reward the better and the lower the Queue Length and Delay the better. Simple Changing Traffic is denoted as 1 in the table and Equally Steady Traffic is denoted as 2. From the plot and table we can see that overall lowering the γ causes the model to perform much worse. The line on each bar is the variability between the values of the configuration and the bar is the difference between those values.

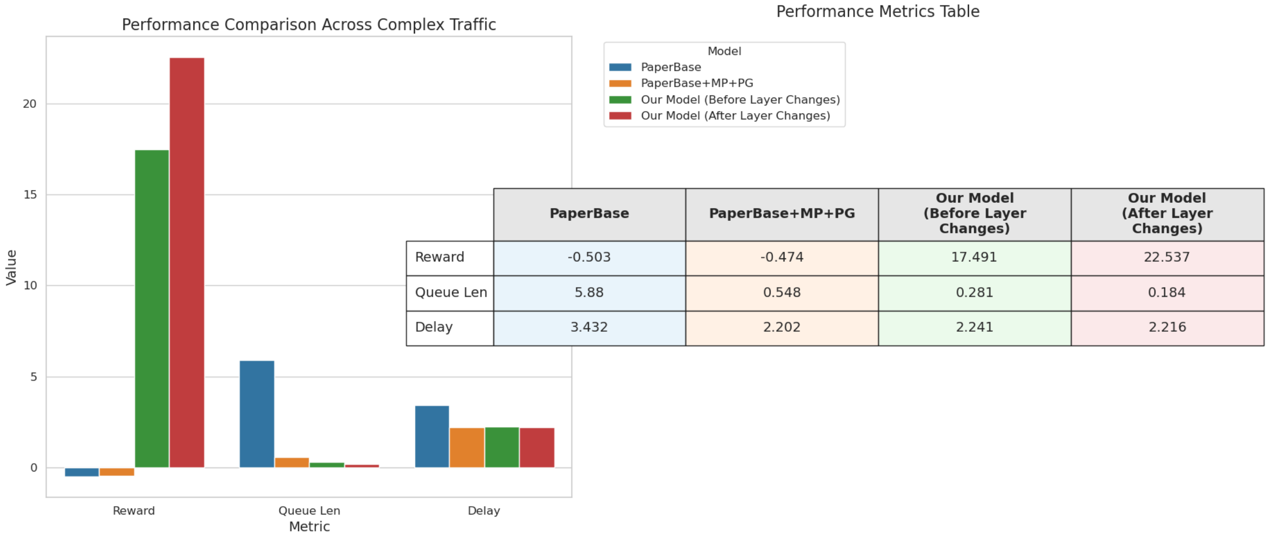
Overall our base model does much better than the results from the paper. So for that reason I will just focus on our implementation results of γ; which are the green and red. As we can see when we change the discount factor to 0.4 the model overall does much worse. The Reward is lower and the Queue Length and Delay are much higher. This is not the results of an optimal model.

We aimed to observe the effect of changing the discount factor and gained new insights about the model. To run both experiments it took four hours. We learned having the agent focus on immediate rewards does not help it find the optimal policy and also does not help the model to converge faster. So even on these simple configurations the model works much better when the agent focuses on prioritizing future rewards.

2.2 Change in Neural Network

Deeper networks tend to outperform shallow networks because deeper network can represent more complex features, allowing the network to be more robust. We wanted to apply this theory to our network and see if we could maximize our reward. For this experiment we used the fourth configuration in the synthetic data set called Complex Traffic. This configuration uses all three previous configurations starting at different intervals throughout the runtime. So the network starts simple and gets progressively more complex as cars start to arrival at a different rate from different directions.

For this experiment we added more Convolutional Layers changing the number of layers from two to four, which we hoped would capture more complex features from the input picture. We lowered the Dropout Rate from 0.3 to 0.2. In doing we so we wanted to reduce the strength of regularization which again can help learn complex features but this increases the risk of overfitting. Lastly, we used LeakyReLU instead of ReLU which allows a small gradient for negative inputs which we hoped would help with learning and faster convergence.



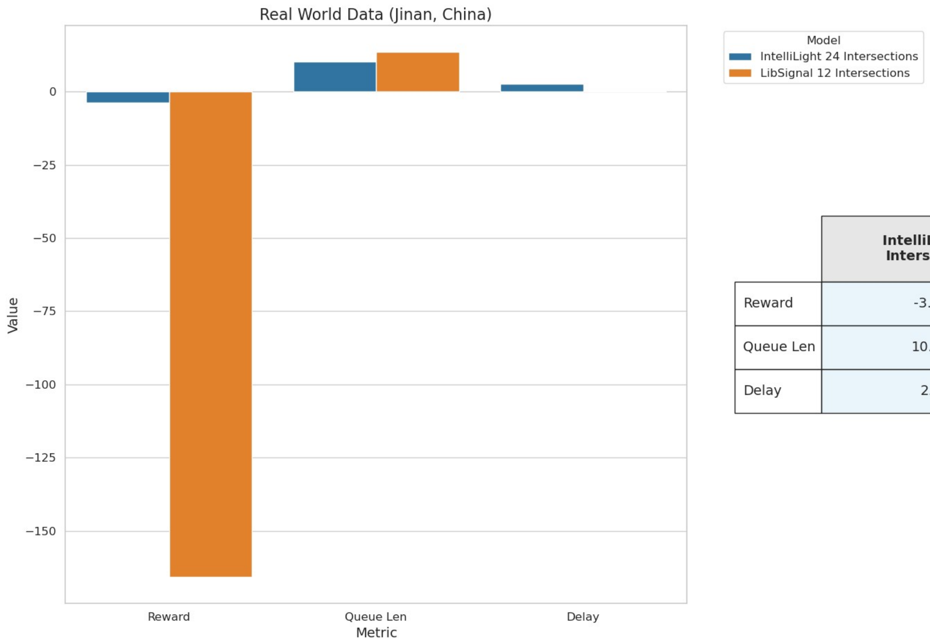
	PaperBase	PaperBase+MP+PG	Our Model (Before Layer Changes)	Our Model (After Layer Changes)
Reward	-0.503	-0.474	17.491	22.537
Queue Len	5.88	0.548	0.281	0.184
Delay	3.432	2.202	2.241	2.216

This experiment took a little longer than four hours to run. Our new model after the changes are shown in red. So to interpret the data the higher the Reward the better and the lower the Queue Length and Delay the better. As we can see from the results, our new model with a much more deeper and robust network was able to outperform all the other models in all metrics. Our new model achieved a higher reward, lower queue length and relatively low delay. Yes, the network took a very long time to run and was computationally costly but it did perform well. We learned that our altered network learned from the data and was able to achieve better performance. So even in complex environments the agent was able to adapt and train well from the data.

2.3 Real World Data

So the initial issue we ran into was the inaccessibility of the real world data that was discussed in the paper which was collected from Jinan, China . This was due to the fact that the authors didn't want to share the real world data due to concerns regarding copyright issues. Then after looking at some of their other work we found that they did publish the real world data in another paper called CoLight, but the data was formatted as JavaScript Object Notation (.json) file. This is an issue for us as the SUMO environment used for our experiment only works with Extensible Markup Language (.xml) files. Even after attempting to convert from one format to another we had no luck. Then after some more digging we found the authors of the paper created a GitHub repository called LibSignal that not only had the Deep Q-Network used in the IntelliLight paper but also had a way to run the .json file via another traffic simulator called CityFlow.

So now we had all the necessary tools to run the real world data. But just a heads up the paper says the Jinan traffic data they used had 24 intersections of traffic but the data available that we used only has 12 intersections of traffic. So the results will vary but we wanted to show and interpret the results as is.



	IntelliLight 24 Intersections	LibSignal 12 Intersections
Reward	-3.892	-165.7086
Queue Len	10.238	13.5889
Delay	2.73	0.2558

So same as before the metric for the data is as follows, we want to maximize the Reward and minimize the Queue Length and Delay. As we have no baseline to compare our data to I will just discuss the results. To run this model it took three hours, which is much faster than the previous experiments we conducted. The reward for our test was significantly lower than any of the results we have seen or reported by the paper. The queue length and delay are actually pretty solid and actually comparable to other models which is a good thing. The reasoning for lower rewards may be due to the environment which it is being trained on or how the model calculates the reward. We don’t know for sure why this is the case as there are too many unknown variability to account for. But we did like the fact that we were finally able run a model with the real world data.

Conclusion

The paper IntelliLight: A Reinforcement Learning Approach for Intelligent Traffic Light Control addresses the traffic light control problem using a well-designed reinforcement learning approach. This was accomplished using both synthetic and real world experiments. We learned how Deep Q-Network can be implemented and how a reward function can be formulated. That a traffic light intersection can be much more complex and involved than we previously understood. We learned that with thorough understanding and creativity of reinforcement learning methodology that these techniques can be implemented in varying environments.

When attempting to replicate the papers results we ran into numerous unforeseen issues and challenges. We knew that the paper and its GitHub repository wasn’t the most up to date but we couldn’t anticipate how poorly documented the repository and python scripts were. The first challenge arose when trying to install the libraries and dependencies as they were extremely outdated. We first used PyEnv to solve this issue but instead stuck with a Conda virtual environment as PyEnv was a slightly intrusive on the on the operating system.

The next issue we had to overcome was running the experiment and figuring out how the average reward was calculated. When the repository is initially downloaded the documentation only states to run the main script and comment out the configuration files not being used. We were never explicitly told that the experiment was only for one small run with no Deep Q-Network being implemented. So we had to go through all the configuration files and adjust the parameter ourselves. We also had to make on python script to average the reward data as the repository had not code that did so.

The last issue we encountered was the lack of real world data for us to use for experimentation. As we discussed previously, we had to go to through many obstacles to not only find the data but also getting a simulation environment to run that data. When we finally found the repository that would run the data, we still had to make a few adjustments to run the experiment. This involved importing the real world data, creating a configuration file for the virtual environment and changes to network to run on the GPU. We were then able to successfully run the real world data.

Applying this reinforcement learning agent to the real world would be extremely difficult. Real-world environments change over time and at times be unpredictable. We would need to consider how to safely explore without risking damage, how to handle limited real-world interactions, and how to design a reward function that truly aligns with desired outcomes. Firstly, designing the correct reward function can be tricky. We simply can’t just add a few parameters and expect to find the most optimal policy. So maybe a more robust agent that excels in continuous action spaces like a traffic intersection with real-value light time phases would do well with Actor-Critic Methods instead. Next, training these agents requires data collection and this can be costly and time-consuming. When implemented in real-world deployment we would need to monitor the agents performance and intervene if needed. Lastly, safety and reliability is paramount. There would need to be safety constraints into the reinforcement learning framework through safety layers and limiting exploration. There are to many unpredictability in the real-world (e.g., traffic patterns, human driving behaviors, weather), so the agent would need to use safe exploration strategies. Applying reinforcement learning to real-world problems demands careful consideration of the complexities of the environment, algorithms and safety. It’s difficult task but not impossible. With the right implementation reinforcement learning to the real-world can be accomplished effectively.