CSCI323 - Modern Artificial Intelligence

FT26 Group Project

A Comparative Study of Model-Free and Model-Based Reinforcement Learning

| Group Member Name | Email | UOW ID |
| --- | --- | --- |
| Jandinero Aylbricht Ramos | jandiner001@mymail.sim.edu | 8761668 |
| Zhao Xian | zxlim017@mymail.sim.edu.sg | 8768845 |
| Pua Yu Xuan | yxpua001@mymail.sim.edu.sg | 8535930 |
| Terence | tjychan001@mymail.sim.edu.sg | 8788352 |
| Keeve Wong | kykwong003@mymail.sim.edu.sg | 8865425 |

Content Page

[**Introduction 2**](#_kajrfnvw59m9)

[**Background Theory 3**](#_n3m6pyz5tq4s)

[Value Iteration and Policy Iteration 3](#_e6e5f6xjhf3h)

[Q-learning 4](#_e1z0io3v6h03)

[Deep Q-learning 5](#_vza79cqwh9lh)

[**Solutions, evaluation, and discussions 6**](#_xxby92lreq0e)

[Value iteration and policy iteration 6](#_ufdgila5pqyy)

[Q learning 7](#_hj2hb2iu7dna)

[Deep Q Learning 8](#_ps3uwvjzp6r5)

[**Conclusion 10**](#_wnho4ng1vwo)

[**Appendix 11**](#_wj6zg19j4qsj)

[**Github Link: https://github.com/zaaoak/CSCI323DriveATaxi 11**](#_f07c5wrlw5qb)

[Acknowledge 11](#_dp1l6uabpsak)

[References 12](#_xrcb3ov7yi7q)

# Introduction

MDPs are a foundational framework for reinforcement learning, it is used to model environments where an agent must make sequential decisions to maximize rewards. In this project, we will investigate model based and model free algorithms in solving the drive a taxi problem made by OpenAI. We will be studying four reinforcement learning algorithms — Value Iteration (VI), Policy Iteration (PI), Q-Learning (QL) and Deep Q Learning DQL— in their ability to learn an optimal policy for gymnasium’s taxi and frozen lake environment.

The Taxi environment is actually deterministic, as each action result in a state with a 100% chance. However our algorithms will still work albeit without the added complexity of probabilities. As our project is just aiming to compare the different Reinforcement Learning algorithms, the deterministic MDP of the Taxi environment would suffice.

The theoretical foundations of this project can be applied to many real life scenarios such as ride hailing, delivery routing and autonomous driving. For example, delivery routing can also be modeled as an MDP as well where companies try to minimize cost and delay.

It is important to note that an MDP is not only about randomness and chances, it's about sequential decision making while maximizing rewards, so for examples such as delivery routing and our Taxi project where it's deterministic, an MDP can still be used. The advantage of using an MDP is that it can be scaled to non deterministic scenarios.

Our focus is on evaluating and comparing these algorithms in terms of convergence speed, computational cost, and the quality of the learned policy. By isolating probabilities, the deterministic nature of the environment allows us to clearly observe how each algorithm behaves under ideal conditions. This also makes it easier to verify the correctness of the implementation and to understand the underlying mechanics of policy improvement.

This project not only demonstrates practical implementation of these algorithms but also provides empirical insights into their performance in a controlled setting. Such understanding is valuable when choosing the most suitable algorithm for real-world decision-making systems where determinism is either present or approximated, such as structured routing problems or robotic motion planning.

# Background Theory

## Value Iteration and Policy Iteration

Traditional Reinforcement Learnings (RL) consists of two different types of dynamical programming methods used for solving the Markov Decision Process (MDP).

Dynamic programming concept was introduced by Richard Bellman in the 1950s, who invented the Bellman Optimality Equation that provides a strong basis of iterative methods like the Value Iteration, where each state of value is updated continuously to estimate the optimal policy (Bellman, 1957). Whereas, Policy Iteration algorithms were then introduced and kicked in by Ronald A. Howard since 1960, to improve the existing Value Iteration by splitting between evaluation and improvement section, benefitting with fewer iterations to converge despite intensive computation required (Howard, 1960).

The main difference relates to its own updating strategies, in which Policy Iteration switches between evaluation and improvement, using to calculate the value function of a fixed policy and update the policy based on the current value function respectively. Because of the full policy evaluation and considerably high computation cost, Policy Iteration converges with fewer iterations. Unlike having the steps of computation and updating, Value Iteration does both procedures in one single update via Bellman Optimality Equation on improving the value function iteratively (Sutton & Barto, 2018). Although Value Iteration can perform simple updates but a drawback of requiring more iterations to converge.

Both RL learning algorithms are similar towards having a key technique in model-based RL learning that can confidently converge to an optimal policy in finite MDP, and current OpenAI Gym's Taxi-v3 environment is one of the examples that convincingly explains and demonstrates how the agent can learn navigation strategies in its optimal way with the help of these iterative algorithms. (Yang, H., 2021)

The main focus of Value Iteration is about updating the state value function with Bellman Optimality Equation iteratively until the value function stabilizes. Though lesser computation per update, Value Iteration requires more iterations to converge. But it can be done in a simpler and effective way such that the given environment is able to be determined and known. Whereas for Policy Iteration is to let the algorithm work by starting off with random policy and to evaluate and improve along the way until convergence happens. Despite having fast convergence on each iteration, it has a drawback of higher computational cost per iteration when running a full policy evaluation. All of these are meant to be learning the efficient navigation sequence and having fewer counts on the common pick-up and drop-off actions in the taxi environment scenario. (Kaelbling, L. P., Littman, M. L., & Moore, A. W., 1996)

## 

## Q-learning

Q- learning history

In 1992, Watkins and Dayan introduced the Q-learning algorithm which uses an incremental method of estimating the Q-function in MDP.

Q- learning theoretical foundation

Q- learning is an off- policy algorithm (Jang, 2019, pg 133656), meaning that the agent continue to update the Q- table with actions that lead to a state with the highest Q- value, Q- table in the self- driving taxi context will be the lookup table that stores the Q- value of all possible (state,action) pair. Even if the action taken is suboptimal due to exploration(e.g. Epsilon = 0.2), the agent will continue to be greedy and take the next action with the best Q- value. This displayed an independent relationship between the learning policy which is used to update the Q- values, and the acting policy (Jang, 2019, pg 133656) which is used to select actions.

In an extreme example, if the agent happen to make exploratory actions(random action) 5 times in a row due to epsilon greedy randomness, and each times end up in a state where Q- value is worse than the previous state, the agent will update(lower) the Q values of each previous states, but the agent will continue to still be greedy and take actions that lead to the best Q- values. From this, we can see that the acting policy remains unchanged. This means that the agent will not avoid making certain actions. However, if an on- policy algorithm were to make exploratory actions “turn left” 5 times in a row, each ending up in a worse state than before, the agent will update(lower) the Q values of each previous states, the decrease in each state’s Q- value will be reflected upon the policy, causing adjustments to be made on policy.

Due to this, the agent will start avoiding the action “left turn”, in turn could lead to many inefficiencies. The second example, on- policy, contrasted off- policy to show the significance of independent acting and learning policy. After convergence, the Q- table is used to help the self- driving taxis to make decisions on actions. At the core, Q- learning has the same Q- value updating idea as the Bellman optimality equation (Jang, 2019, pg 133656), bootstrapping future rewards, which means updating future rewards on the fly using the current Q- table(current knowledge). Q- learning is also model- free like Monte Carlo methods (Jang, 2019, pg 133656), they both learn from experience, update estimates(Q- values) from trying actions and observing what happens, not requiring any prior knowledge of the environment’s dynamics.

While Q- learning has great learning ability in a single- agent environment, it struggles with complex problems with many (state,action) pairs. This struggle is due to the Q- table scaling exponentially with the (state,action) pairs, in turn making it computationally expensive (Jang, 2019, pg 133656) to store the Q- table. Q- learning will also converge slowly (Jang, 2019, pg 133656) because it only updates one state at a time(one cell in the Q- table), and will rarely visit many states causing those states to update even slower. While all of this could be improved by having multiple agents, the basic Q- learning simply cannot work with multiple agents, because multi- agent cause the Q- table to be outdated as the other agent updates, and the rewards change unpredictably due to having more agents updating at the same time.

## Deep Q-learning

**Brief history**

Deep Q-Learning was popularized in 2015 when researchers at DeepMind combined reinforcement learning with deep neural networks to create a system that could learn to play Atari games from raw screen pixels — no manual programming needed.

This approach, called the Deep Q-Network (DQN), marked a major step forward because it showed that AI could learn complex tasks directly from high-dimensional input like images, using only rewards as feedback. It helped launch the modern era of deep reinforcement learning.(Mnih et al. 2015, pp. 529–533).

Deep Q-Learning emerged as a powerful solution to the challenge of applying reinforcement learning in environments with complex, high-dimensional inputs. Traditional Q-learning methods struggled with scalability because they relied on discrete state-action tables, which become impractical as the number of states grows. The integration of deep neural networks to approximate Q-values allowed agents to generalize from limited experience and handle continuous or very large state spaces. (Li 2018, pp.5-6)

**Usage**

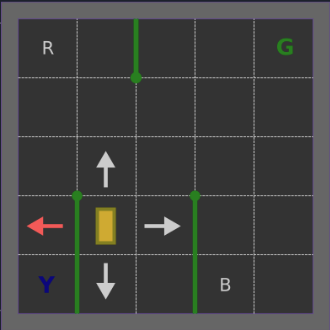
Q-Learning uses the Q-table to help the agent decide which action it should take. In an environment where it is not as complex and low in dimensionality, the Q-table is small and manageable to work with. However when the environment is higher in dimensionality, the Q-table gets exponentially larger and harder to manage. This is where Deep Q-Learning can help, replacing the Q-table with a neural network that can approximate Q-values for a continuous or high state space such as an image. Resulting in a model that learns patterns and generalizations without having to identify all possible states. This makes Deep Q-Learning more applicable to real-life situations where it is able to handle the enhanced dimensionality.

The Neural network architecture is as follows, the total number of nodes in the input layer corresponds to the total number of features and the output layer corresponds to the number of possible actions the agent can take. Each output node represents the estimated Q-value for taking a specific action in the current state, allowing the agent to choose the action with the highest predicted value.

To improve model performance, Deep Q-Learning integrates experience replay, where records of the agent’s actions and past experiences are stored in the replay buffer, mini-batches of these past experiences are then randomly sampled to train the network

# Solutions, evaluation, and discussions

Python scripts were developed to simulate the MDP environment and run each RL algorithm. Excel sheets were also used to manually calculate the expected value updates over time. We used PyTorch for the neural network for Deep Q Learning and MatPlotLib for plotting graphs to visualize results.



In the gymnasium Taxi environment, a passenger can be in 5 states, R G B Y or in Taxi, the taxi can be in 5 different columns and 5 different rows, the destination can be in R G B or Y, this gives us a total of 5 columns \* 5 rows \* 5 Passenger Locations \* 4 destination = 500 states, as a passenger cannot start at the same destination, we do not have to consider taxi locations when passenger is at the destination, so it will be 400 reachable states, and since passengers being dropped off at the destination is another state, the terminal state, that is a total of 404 reachable states.

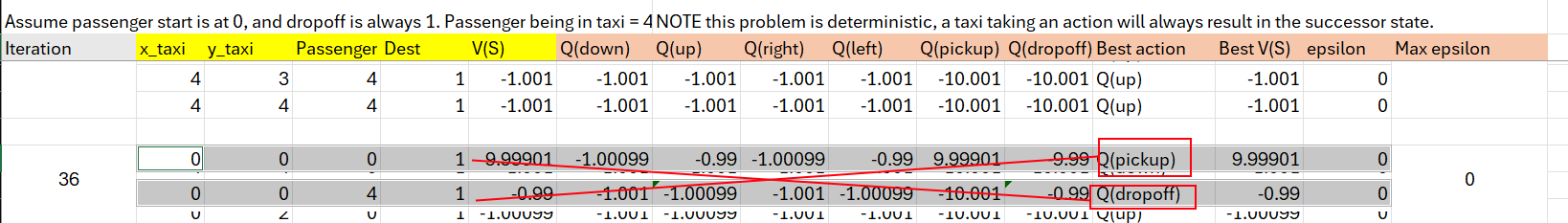
However, we will only be testing 51 states for value iteration and policy iteration, this is because we want to keep it simple for our calculation to understand the algorithms, our passenger start state will only be set to the red square, and the destination will be set to only the green square, this gives up only 5 columns \* 5 rows \* 2 passenger locations(pickup and in taxi) \* 1 destination + 1 terminal state(drop off) = 51 reachable states.

The rewards for this MDP are -1 for every step taken including picking up passengers, -10 for doing an illegal pickup or drop off, and +20 for dropping off at the correct location. It is important to note that, dropping off passengers at an incorrect destination will change the state to where the passenger is at the hotel waiting to be picked up, there won't be a -10 penalty for this, but rather just a -1 time step penalty. Dropping off a passenger in the middle of nowhere will not change states at all, but get a -10 penalty.

Some might look at the rewards and wonder why we are not given a reward for picking up the passenger, but we will see why it is important to model rewards carefully.

## Value iteration and policy iteration

Using value iteration, if a reward of 10 is given to the agent for picking up a passenger, when the values have converged after training, you will see an infinite loop where the agent learns that picking up and dropping off a passenger at the same location will give it the most reward over driving the passenger to the correct destination. This shows the importance of modelling rewards properly!



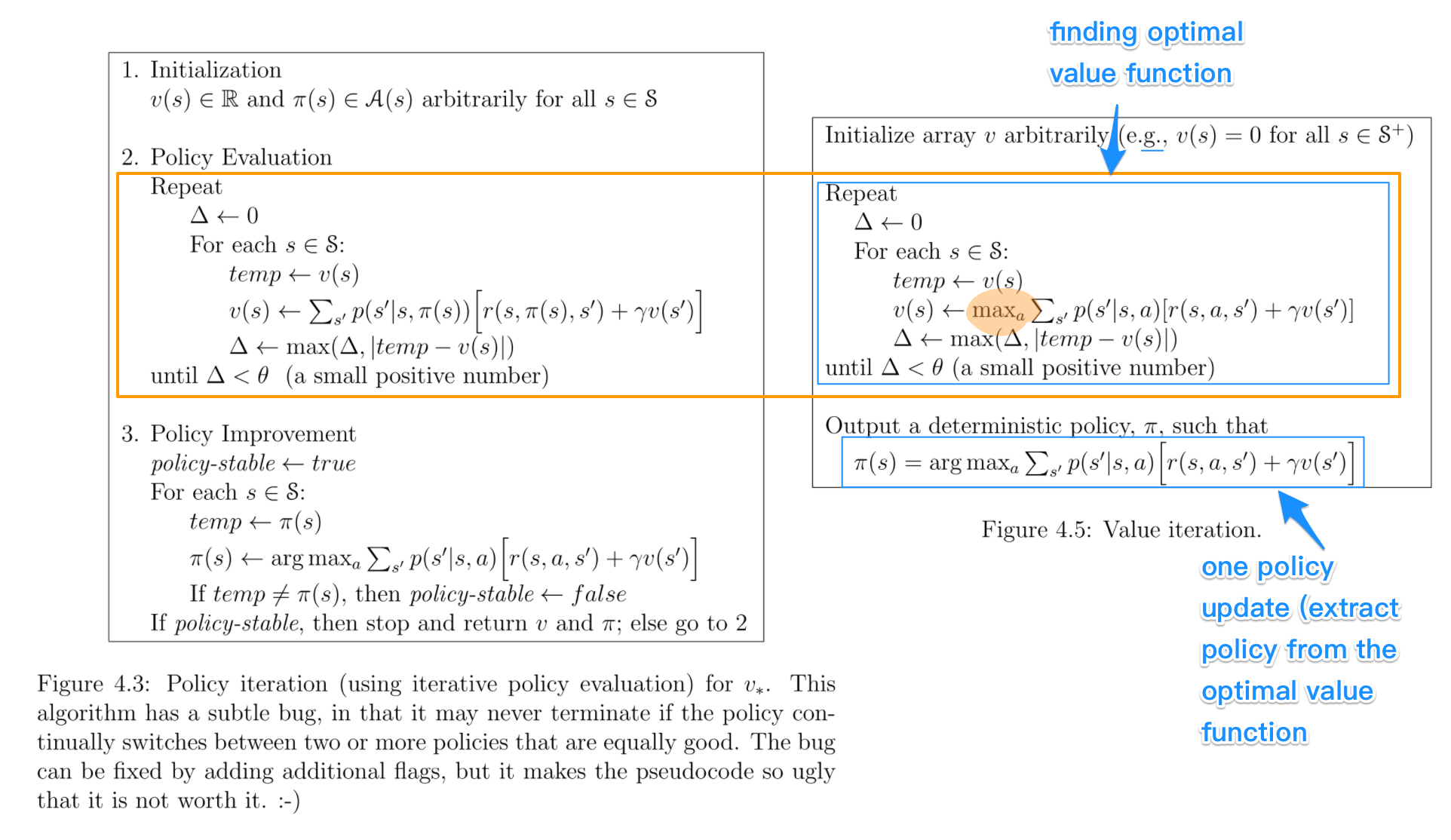
After shaping the rewards properly, it was noted that the value iteration algorithm converges at around 19 iterations, where epsilon is 0, note that epsilon converges to 0 because of the fact that the taxi environment is deterministic, the rapid convergence is also attributed to the fact that our MDP is very simple and deterministic.



For policy iteration, it has converged after 17 iterations (epsilon = 0), but it should be noted that, it took more computations to reach 17 iterations, steps here represents everytime a policy evaluation step is taken.



The difference can be explained in this pseudocode provided by Sutton and Barto's book: *Reinforcement Learning: An Introduction*



(Sutton & Barto, 2018, pp.75, 80)

In the figures above, you can see that Value Iteration converges to V\* by directly applying the Bellman optimality equation iteratively, it improves the value function each step and derives the policy from it.

Policy Iteration converges to V\* indirectly by iterating over policy evaluation and policy improvement until the policy becomes stable. It improves the policy and then computes the exact value function under that policy.

In theory policy iteration has lesser iterations, but each iteration takes more computational power as can be seen in our python code where there were 58 sweeps of value function calculations.

## Q learning

We used the frozen lake environment for Q learning and Deep Q learning, as the frozen lake environment can be modified to only have 16 states, allowing us to model it for Deep Q learning much easier.

Frozen lake gives a reward of 0 for each step and falling into the hole, which is a terminal state as well. While reaching the goal gives a reward of 1. We set the slippery flag to false in this environment, which means that the problem is deterministic similar to our taxi environment.

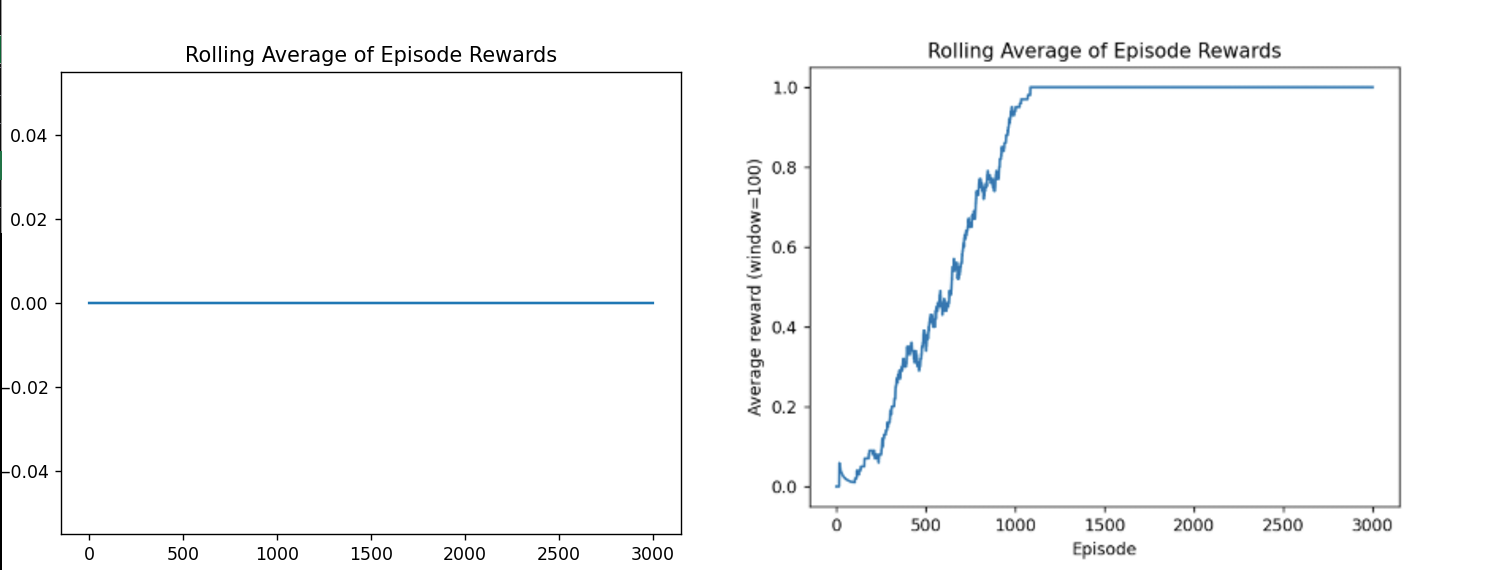


Q learning updates the tabular data one (state,action) pair at a time based on action sampling, so it is crucial that we implement an exploration-exploitation tradeoff which is where epsilon-greedy comes in. Every time it samples an action, there is a chance that it will take a random action or the best action from Q-table, depending on the value of epsilon.

We have set up the environment with a learning rate of 0.01, discount factor of 0.9 and epsilon decay of 1/1000.

We tried 2 epsilon greedy policies, on the left is where it decays to 0 after 200 episodes, meaning it does pure exploitation after that, and on the right is where it decays to 0 after 1000 episodes.

As you can see from the graph, if epsilon decays too fast and pure exploitation happens too early, the Q values will not be updated optimally,because when pure exploitation happens, the agent will only choose the “best” action according to the Q Table, which might be a very bad action due to the Q Values not being optimal yet.



Even though with an epsilon greedy strategy, some state action pair may be sampled less than others and their Q values may not be optimal at certain points of the training, there are proofs that Q learning algorithm will always converge to an optimal value, so long as each state,action pair are sampled an infinite number of times, the rewards are bounded within an infinite size, and the learning rate must decrease infinitely. (Watkins & Dayan 1992, pp.57-58)

## Deep Q Learning

We implemented deep Q learning in frozen lake with learning rate= 0.01, discount factor of 0.9, an epsilon decay rate of 1/1000 for the epsilon greedy algorithm. We also set the experience replay to 1000, mini batch size to 32 and network sync rate to 10. We will train for 3000 episodes.

Network sync rate is when the target network syncs to the policy network, during training, our target network is frozen while policy network is being trained. This is important because having a moving target network, that is a target network that is continuously synced to the policy network, would cause instability and poor training for our policy network, think of it like training to shoot but the target keeps moving around.

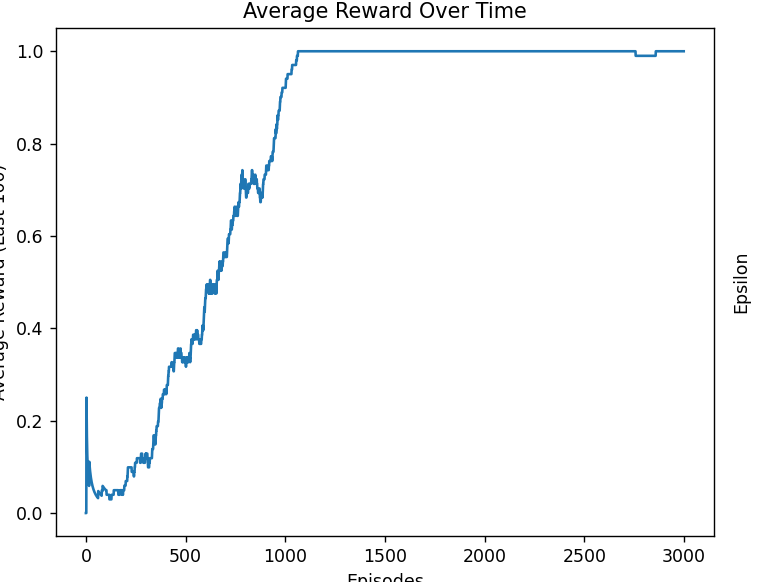
Experience replay is when the algorithm stores past experiences in a buffer, and after a certain number of steps, takes a random mini batch from it to train the policy network, it is another important feature because it breaks correlation in sequential data, this is similar to overfitting. An analogy is that a sports coach only coaches you based on the last training session, instead of over a range of previous training sessions, this leads you to overfit to only the previous training session, even if the previous training session only focused on one part of your body.

Immediately we noticed that it takes much longer to train due to the overhead of a neural network which requires a forward propagation of the policy and target network, and a backward propagation of the policy network every step it takes in the episode. In contrast to Q Learning where each state-action pair Q values have a straightforward and fast calculation, DQL involves calculation gradients and updating weights through back propagation which is computationally intensive, below is a comparison of the average time to train per episode for both Q learning and deep Q learning over 3000 episodes each.





After epsilon decays to zero around episode 1000, the agent exploits the learned policy fully. However, rewards sometimes drop afterward even if the policy is already taking all the right steps because DQN uses function approximation, which can lead to occasional errors in estimating Q-values. Since the policy network trains continuously against the target network, these approximations may cause the agent to choose suboptimal actions occasionally, resulting in temporary performance dips.



# 

# 

# Conclusion

Throughout the project, we implemented model-based and model-free reinforcement learning algorithms and gained a deeper understanding of their behaviour. One of our key takeaways are that value iteration and policy iteration have to be carefully considered before implementation as they take different approaches to solve a problem, even though policy iteration converges in fewer iterations, there is overall more complexity involved due to the policy evaluation in each iteration, thus it is very important to weigh the trade offs before deciding on a model based algorithm.

On the other hand, problems where we don't know the dynamics of the environment can be solved using model-free techniques like Q-learning and Deep Q-Networks (DQN). Without the use of a transition model, these algorithms are able to approximate optimal behavior by learning directly from interactions with the environment. Putting Q-learning into practice made it clear to us how straightforward yet effective the algorithm is, particularly in isolated settings like FrozenLake. But it also brought attention to issues like the necessity of parameter tuning and trade-offs between exploration and exploitation. We gained practical experience with deep reinforcement learning methods such as function approximation, experience replay, and target networks after switching to DQN, which added complexity but also increased generalization potential.

A major insight from the project was that there is no best algorithm in reinforcement learning. The choice of algorithm depends on the environment's properties, computational resources, and the available knowledge about the environment.

Beyond the algorithms themselves, this project has increased our confidence in our ability to visualize agent behavior, implement complex learning systems, and critically interpret training results. It also reaffirmed how important it is to comprehend not only how algorithms operate but also why they behave in particular ways depending on the situation.

# 

# Appendix

# Github Link: <https://github.com/zaaoak/CSCI323DriveATaxi>

## 

## Acknowledge

| Name | Contribution | Contribution Percentage |
| --- | --- | --- |
| Jandinero Aylbricht Ramos (8761668) | -Code | 100 |
| Zhao Xian (8768845) | -Code  -Solutions | 100 |
| Pua Yu Xuan (8535930) | -Background theory Value iteration and policy iteration | 100 |
| Terence (8788352) | -Background theory q learning | 100 |
| Keeve Wong (8865425) | -Background theory Deep q | 100 |

## 

## 

## References

B. Jang, M. Kim, G. Harerimana and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," in IEEE Access, vol. 7, pp. 133653-133656, 2019, doi: 10.1109/ACCESS.2019.2941229.

keywords: {Reinforcement learning;Mathematical model;Classification algorithms;Machine learning algorithms;Market research;Reinforcement learning;Q-learning;single-agent;multi-agent},

Mnih, V, Kavukcuoglu, K, Silver, D, Rusu, AA, Veness, J, Bellemare, MG, Graves, A, Riedmiller, M, Fidjeland, AK, Ostrovski, G, Petersen, S, Beattie, C, Sadik, A, Antonoglou, I, King, H, Kumaran, D, Wierstra, D, Legg, S & Hassabis, D 2015, *Human-level control through deep reinforcement learning*, *Nature*, vol. 518, no. 7540, pp. 529–533, viewed 20 May 2025, <https://doi.org/10.1038/nature14236>.

Li, Y 2018, ‘Deep reinforcement learning: An overview’, *arXiv preprint*, arXiv:1701.07274, pp5-6, viewed 20 May 2025,<https://arxiv.org/abs/1701.07274>.

Watkins, C. & Dayan, P. 1992, ‘Q-learning’, *Machine Learning*, vol. 8, no. 3–4, pp.57-58.(<https://link.springer.com/article/10.1007/BF00992698#preview>)

Yang, H. (2021). An independent study of reinforcement learning and autonomous driving. arXiv. <https://www.researchgate.net/publication/355367571_An_Independent_Study_of_Reinforcement_Learning_and_Autonomous_Driving>

Bellman, R. (1957). Dynamic Programming. Princeton University Press. <https://www.worldcat.org/title/dynamic-programming/oclc/181368>

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Value iteration. In *Reinforcement learning*: *A survey. Journal of Artificial Intelligence Research, 4, 237–285*. Retrieved from <https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node19.html>

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Policy iteration. In *Reinforcement learning*: A survey. *Journal of Artificial Intelligence Research*, 4, 237–285 Retrieved from <https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node20.html>

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press., 80*–*87 <http://incompleteideas.net/book/the-book-2nd.html>

Tokuç, A. A. (2025, February 13). *Value iteration vs. policy iteration in reinforcement learning*. Value Iteration vs. Policy Iteration in Reinforcement Learning | Baeldung on Computer Science. <https://www.baeldung.com/cs/ml-value-iteration-vs-policy-iteration>