CS7641 A4

Markov Decision Processes

Wei En Chen

***Abstract* – This Paper demonstrate three different algorithms, value iteration, policy iteration, and Q-learning on two different RL problems. When state transitions are known, VI and PI can quickly find global optimal solution. When state transitions are unknown and the states are continuous, DQN (Q-learning's extension) is very robust in finding a very good solution.**

I. INTRODUCTION

MDP (Markov Decision Process) is a decision-making process that allows some stochasticity and control. An MDP follows the Markov property which says the prediction of the next state can be done by the current state entirely. In other words, an MDP is memoryless, in the sense that it does not keep an explicit record of the entire history of the decision-making process [1,2]. The main difference between MDP (or reinforcement learning) and supervised learning is that for supervised learning, we are given input x and output y, and we figure out f(x) to predict y (as y hat). In reinforcement learning, we are not mapping directly from input x to output y with a f(x), instead we are given multiple intermediate states by interacting with the environment before reaching the goal, like playing chess, where each player moves pieces from states to states and the goal is to win the game at the end. This makes supervised learning very difficult to learn.

Typically a reinforcement learning (MDP) problem has an agent, which can learn to make optimal (global or local) decisions. It also has an environment which the agent lives in. As far as the agent is concerned, the environment is the entire world. For chess, it's the 8 x 8 grid chess board with all the pieces and all the rules. MDP also includes actions that allows the agent to move from one states to another. In chess, the actions are deterministic, and they can be moving a rook from one position to another as long as it follows the rules of the environment. The states represent positions of all pieces on the chess board. The actions can effectively move from one state to the another. Another important item using chess as the example here is the policy, which tells the agent what piece to move in any particular states. The last thing is the reward. In chess, the only obvious reward is at the end of the game, win or lose. This delayed reward is typical in reinforcement learning, and it is a big reason why reinforcement learning is more difficult to learn compared to supervised learning.

In this assignment, we will look at two different MDP problems, and apply three different algorithms to solve the problems. Behaviours, pros, and cons will be discussed for each algorithm.

II. MDPS

FROZEN LAKE

This is one of the most popular grid-world (inherently discrete states) problem for MDP offered by OpenAI Gym [3]. The environment is a square map and the starting point is at the top left corner, and the goal position is at the bottom right corner. The rest of the grids can be either a frozen land, or frozen lake. A character starts at the top left grid, and should navigate through the map to the goal position without stepping onto the frozen lake, which will end the episode. The observation state is the location of the character in the map, and the allowed actions are left/down/right/up, both represented by integers. I find this problem interesting because a slippery state can be enabled such that 1/3 of the time the next state is the desired state from the action, and 2/3 of the time the character will end up in perpendicular direction of the applied action. In addition, the size of the map can be adjusted which will be interesting to observe how algorithms perform. Figure 1 shows a snapshot of the 4x4 frozen lake problem.

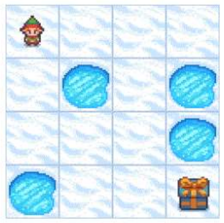


Figure 1 - Frozen lake (from gym) map 4x4 visualization.

CART POLE

This problem is also from OpenAI Gym [4]. A cart can be moved left/right as 1D motion while a pole is pinned on the cart. The goal here is to balance the pole such that it is always pointing up by moving the cart left/right for 500 steps. A snapshot of the cart pole problem is shown in Figure 2. The observation state consists of cart's position, cart's velocity, pole angle, and pole angular velocity. The action is simple which is to move the cart left/right represented by integers. Unlike frozen lake, the state space here is continuous, and there are 4 numbers. I find this interesting as the continuous state space is a big contrast and it likely will highlight performance differences among RL algorithms.

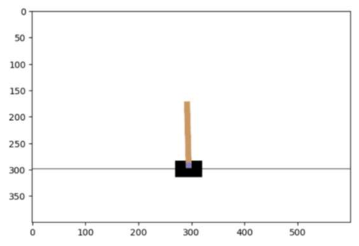


Figure 2 - Cartpole RL problem (from gym) visualization.

III. VALUE ITERATION (VI)

Value iteration utilizes Bellman equation to calculate the "utility" in each state. The utility function looks something like the following [5]:

$$U(s) = R(s) + \gamma \max_a [\textstyle\sum_{s'} T(s,a,s') U(s')] \quad (1)$$

Where R is the reward for being at state s, γ is the discount factor, T is the transition probability from state s, taking action a, and end up in s'. We can see a couple characteristics of this function. First, U is on both sides of the equation with different states, so this is inherently is a recursive computation, hence value "iteration". Second, γ is applied to the future utility which means it determines how the agent perceives the value of future actions.

Dynamic programming method was developed by Bellman [6] and is used to solve the optimization of the values at each state. It works by initialize each state with random utility values, and then update each state utility values using Equation 1, and then repeat the update until utilities don't change, or the change in utilities is smaller than some threshold from previous iteration to save some computation. To find the optimal policy, simply follow the utility in the manner that the chosen action will maximize utility transition from s to s'. Sometimes a Q table is generated to store utility of every state and action pairs, and the best action is simply the action that gives the highest utility at a given state.

Applying VI to frozen lake should be relatively easy since the transition probability T is well known, deterministic (not slippery) or probabilistic (slippery). In contrast, applying VI to cart pole problem will face some challenges, such as the number of states are effectively infinite because of the floating values. It can be solved by discretize the range into discrete number of bins. We can quickly see the number of states can easily blow up in this case. There are 4 floating values, and if each value is discretized into 10 zones, we have 10^4 states. If we double the zones, the number of states is increased by 2^4 times. In addition, the state transition probability is unknown. Sampling method is then required to estimate the transition probabilities.

IV. POLICY ITERATION (PI)

Policy iteration is also a recursive method to optimize the policy. It works by first initializing a random policy, that is, randomly choose a "best" action at each state, and then evaluate the utility at each state using the following formula [7]:

$$U_t(s) = R(s) + \gamma \sum_{s'} T(s, \pi_t(s), s') U_t(s') \qquad (2)$$

This step is typically called policy evaluation. Compared to value iteration, there is no "max over action" term because action is dictated by the policy at iteration t. Once the utilities are updated according to the current policy, the utilities are used to update the policy by [7]:

$$\pi_{t+1}(s) = \underset{a}{\operatorname{argmax}}[R(s) + \gamma \sum_{s'} T(s, a, s') U_t(s')] \qquad (3)$$

This portion is called the policy improvement. It essentially looks at every state and find the action that produces the maximum utility. Repeat policy evaluation and policy improvement until convergence, that is, no more changes in policy. Note that the convergence criterion is different from VI. Same as VI, if the states are floating values, discretization and some sampling method are required to estimate the state transition probabilities. They share the same fate that the number of states can easily blow up when there are multiple observable variables.

To deal with continuous-state RL problem, we will need to look at a more powerful algorithm, which is introduced next.

V. Q-LEARNING and DQN

My favourite RL algorithm is q-learning and its extension, deep q-learning (DQN). It is an off-policy learner. Off-policy means the learned Q, state-action pair function, is independent of the policy being followed [8]. The following shows the main Q-learning update equation:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \qquad (4)$$

It typically follows some ε-greedy policy with reducing ε, shown by the max over a. This behaviour means its "online" performance during training would be lower than on-policy learners such as SARSA [8]. However, what really matters is the final learned model and the advantage of Q-learning is that it can use past experience as training examples to learn because as stated before, the learned policy is independent of the policy being followed. The past experience replay can be very useful as we move from discrete to continuous state realms. In the continuous state realm, an advanced Q-learning algorithm is introduced as deep Q-learning networks (DQN).

There are a few very important key features in DQN to make it work besides the past experience replay. The "networks" in DQN means there are neural networks involved. The reason we need this is that the agent knows nothing of the world it is in, and there is no Q table that prescribes the state-action pairs. Q then can be estimated by neural networks, which are essentially estimators [9]. To train a neural network, we need data and this is where past experience replay becomes useful.

However, there is a big problem with simply sticking a neural network in Q-learning. In supervised learning, input x and ground truth labels y are given. In RL, there are no given ground truth labels. Instead, we have the best estimated Q at each time step, meaning the target is changing as it learns more about the world. This is usually disastrous for neural networks training because the "ground truth labels" are now moving targets. NNs cannot properly learn via gradient descent when gradients keep changing, which affect the weights, which results in instability. To solve this issue, a t-soft update is utilized [10]. In a nutshell, it uses a hyperparameter, τ, between 0 and 1, to slow down the weight updates of the network we are trying to estimate the best Q. This method improves the stability of the training [10].

The downside of DQN is, as one can imagine, more hyperparameters to tune, and more computation time as it involves experience replay, and it does not guarantee convergence to global optimal solution, since neural networks are estimators. However, it is likely still better than using VI or PI with discretized states, which is also a type of estimation, which we will show in the analysis.

VI. ANALYSIS

FROZEN LAKE

All three algorithms to train under frozen lake are from bettermdptools with some custom modification [11]. First we look at the 4x4 frozen lake problem, illustrated in top left of Figure 3. S is start, F is frozen land, H is hole, and G is goal. When VI, PI, and QL are used to solve this puzzle with deterministic state-action transition, the state values of the entire map are shown in the rest of the Figure 3. Top right and bottom left are VI and PI, respectively, with gamma=0.99. We can see they both converge to the same values. For QL, we can see that the state values are not the same, especially on the states not on the optimal path. The reasons are VI and PI use the underlying state transition model and the dynamic programming will update all possible states until all states come to equilibrium. For QL, it knows nothing about the state transitions and it only learns from actually playing the game. Since the starting position is always fixed, there is some chance QL agent may not visit all states enough times while using exploration-exploitation method.

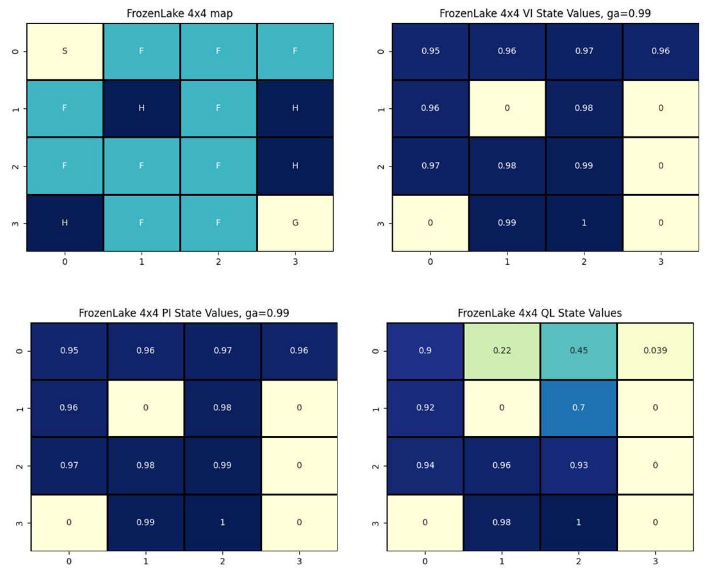


Figure 3 - Converged frozen lake utilities with ϒ=0.99. TopLeft: map. TopRight: value iteration. BottomLeft: policy iteration. BottomRight: QL

For small map size, QL (model-less) may end up with the same optimal path from start to goal compared to VI/PI (model-based). By having a larger map will quickly show the optimal path can be different, in Figure 4, due to inherent randomness in QL. Bottom right is the optimal path by QL, and we can see that it is different from others. In deterministic world, gamma would only affect the converged state values with the same exponential rate. With larger maps size and low gamma, the state values would be closer to zero on the states far away from the goal. If the convergence tolerance is not appropriately assigned, the low gamma could produce a different global policy. In probabilistic state-transition, this can change because gamma is acting as a risk-taking parameter. An example is shown in Figure 5. On the left is the VI global policy with gamma=0.99, and on the right is the VI global policy with gamma=0.8. The map is the same as the previous 8x8 map. We can observe that at states (row:1, col:1 and 2), the global policy with gamma=0.99 wants to take less risk by directing the agent to go up, which is away from the hole in state (row:2, col:3). With gamma=0.8, the reward diminishes faster so the global policy directs the agent to go the right until it is very close to the hole.

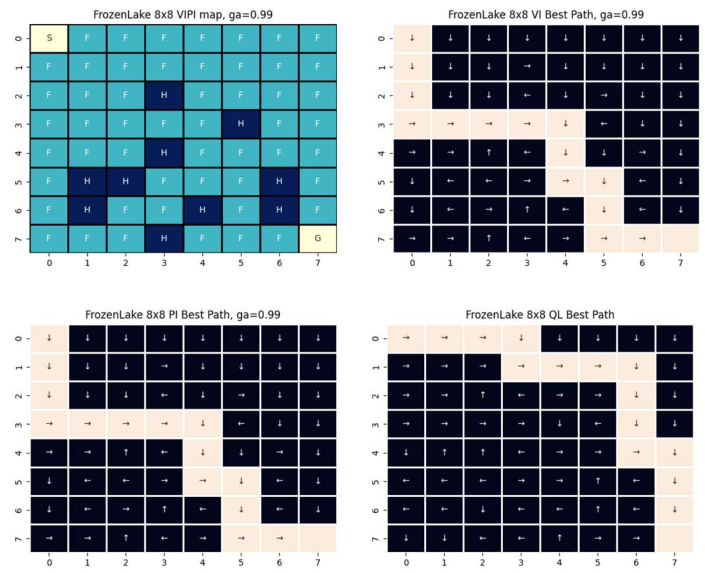


Figure 4 - Converged frozen lake best path with ϒ=0.99. TopLeft: map. TopRight: value iteration. BottomLeft: policy iteration. BottomRight: QL

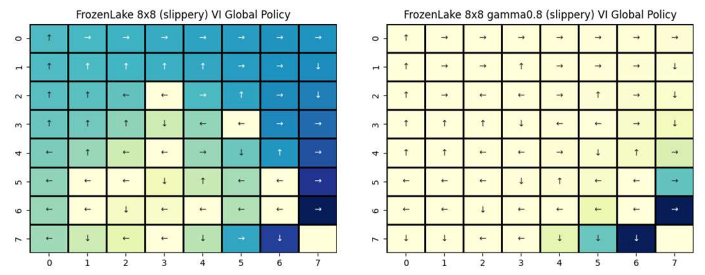


Figure 5 - Effect of discount factor, ϒ, on global policy. Left: ϒ=0.99. Right: ϒ=0.8

Let's look at the overall behaviour of VI and PI in deterministic state transition world. Plots are summarized in Figure 6. Each curve represents a gamma value. Multiple runs were executed to have better understanding of the mean and variance. The top 2 plots show the number of steps required to converge, and they are the same between VI and PI. This makes sense because in each iteration, the score of 1 at the goal is propagated out by exactly one closest neighbour. It will take the same amount of iterations to propagate the score throughout the entire map. Gamma makes no difference for map size 16 and below because the station transition is deterministic. Note that for map size of 32 with gamma=0.7, convergence steps is no long in linear relationship by VI, which is different from PI. This behaviour is due to low discount factor diminishes the values closer to zero such that VI believes convergence is achieved based on a given threshold. For PI, as mentioned earlier, there is a policy improvement portion which checks for any changes in policy. This additional check will force the algorithm to do more updates and therefore the convergence is still in linear relationship to the map size.

As for the 2nd row of the plots, we can see the wall clock time for VI is a lot faster than PI, especially when map size increases to 32. PI takes longer because each policy iteration step includes value evaluation which is also iterative. The wall clock time curve is exponential due to number of states are quadrupled when map goes from 16 to 32.

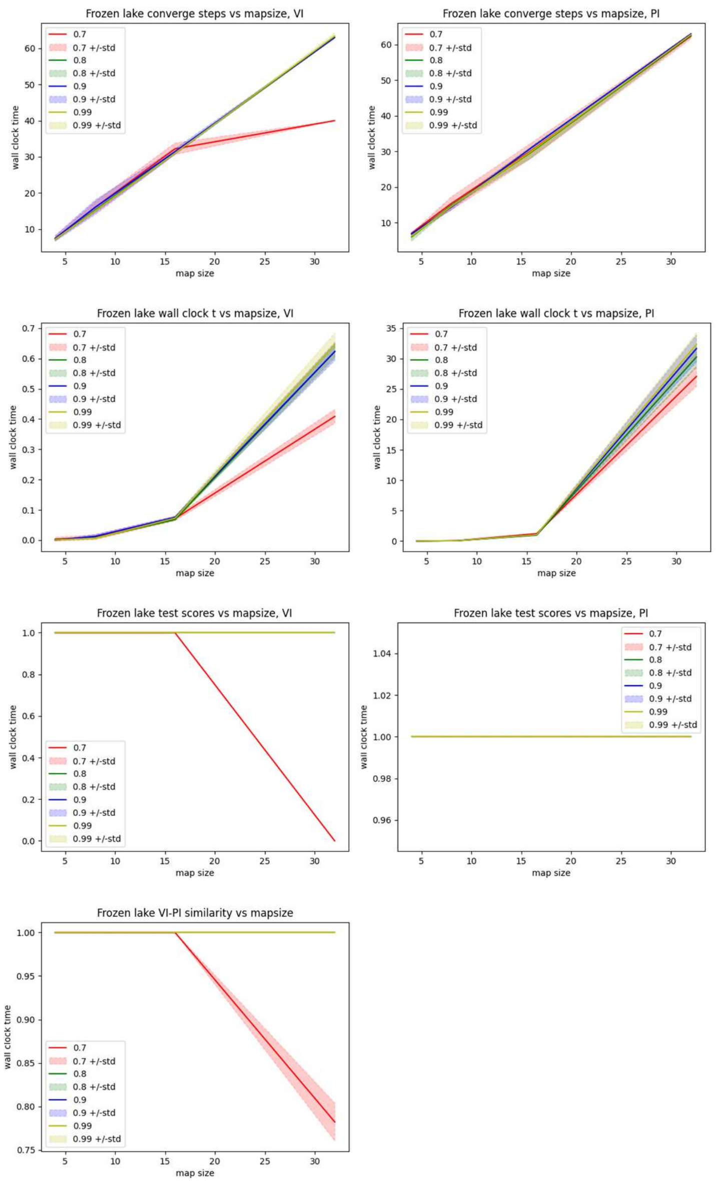


Figure 6 - VI PI Performance vs map size on deterministic state transition. 1st row: steps. 2nd row: wall clock time. 3rd row: average test scores. Bottom: VI vs PI utility matrix similarity.

The 3rd row plots show the test scores where the trained model goes through multiple games and get the mean. Since there is no stochasticity, no variance is expected. The score of VI at map size is 32 and gamma=0.7 goes to zero because the discounted reward close to the start is lower than the set convergence threshold so it did not properly converge like PI. An even smaller threshold is needed when situation like this occurs.

The bottom plot simply compares the similarity of states between VI and PI by counting states with the same utilities. As expected, some of the states do not match with map size of 32 and gamma=0.7.

In the world of stochastic state transition, summary plots are shown in Figure 7. The top row plots are the wall clock times. Similar to deterministic case, they are exponential to the map size. However, higher gammas require longer time to converge in both VI and PI, and they are even longer than deterministic case. This is due to extra calculations from summing probabilities of ending in different states after applying an action. The low gamma cases converged a lot faster here because the discounted reward simply could not reach far enough to the start state. We can observe the test scores in the bottom row of plots. With low gammas, the scores quickly dwindle down to zero as map size increases to 16 in both VI and PI. With gammas of 0.99 and above, the test scores can still manage to get to goal without stepping on holes. The reason that the agents cannot 100% get to the goal is because of the stochasticity and the maps may have narrow paths with surrounding holes.

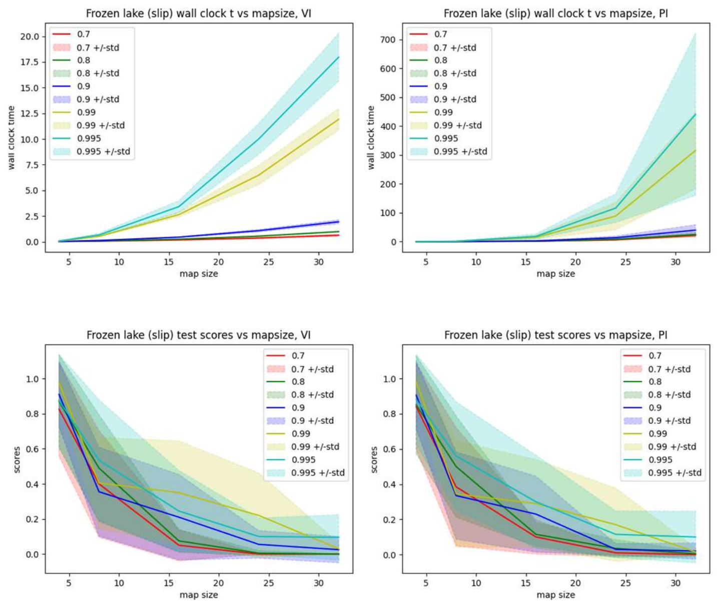


Figure 7 – VI PI Performance vs map size on probabilistic state transition. TopRow: wall clock time. BottomRow: average test scores.

Next, we can look at QL algorithm hyperparameter performance on the frozen lake (deterministic only). To help with faster convergence under QL, some reward shaping is done such as a reward of -0.01 is applied in each action the agent takes. This has the similar effect as gamma but it provides constant feedback to the agent instead of the final reward at the end of the episode if it reaches the goal. Another reward shaping is -1 when agent steps on the hole. Reward shaping definitely alter agent's behaviour towards either risk-averse or not. However, they can also provide more feedback to increase convergence speed. For convergence criteria, if the agent is able to get to the goal 10 consecutive times, it is considered convergence has been met, or maximum episodes to train is 70k.

For gammas as a hyperparameter, we can see some result plots in Figure 8. We can see that when gamma is low, the time and steps it takes to convergence is a lot longer and the average reward is lower. Low gamma results in short-sightedness so the agent may find it more difficult reaching the goal. The best gamma for map size of 8 is likely around 0.7. Note that even at gamma=0.7, sometimes the agent still cannot successfully reach the goal, which is due to stochasticity within QL during learning that it can sometimes get stuck at local minima especially if the hyperparameters are not well tuned.

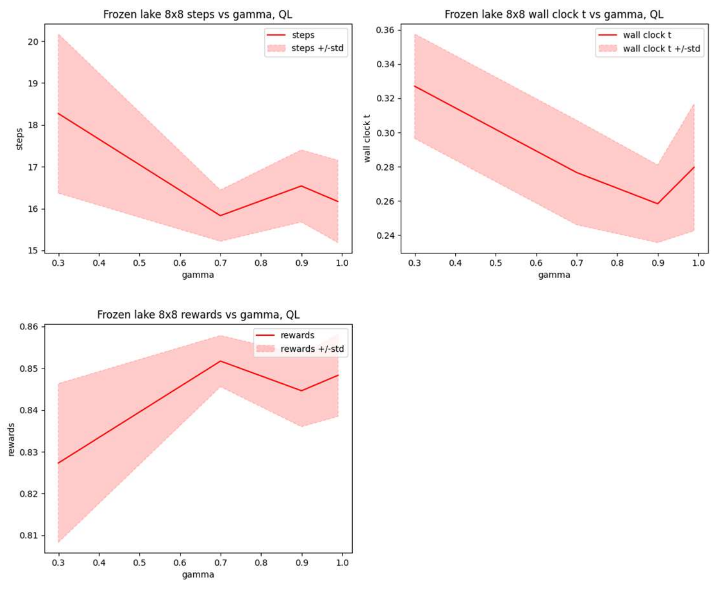


Figure 8 – Effect of ϒ on QL. TopLeft: steps. TopRight: wall clock time. Bottom: rewards.

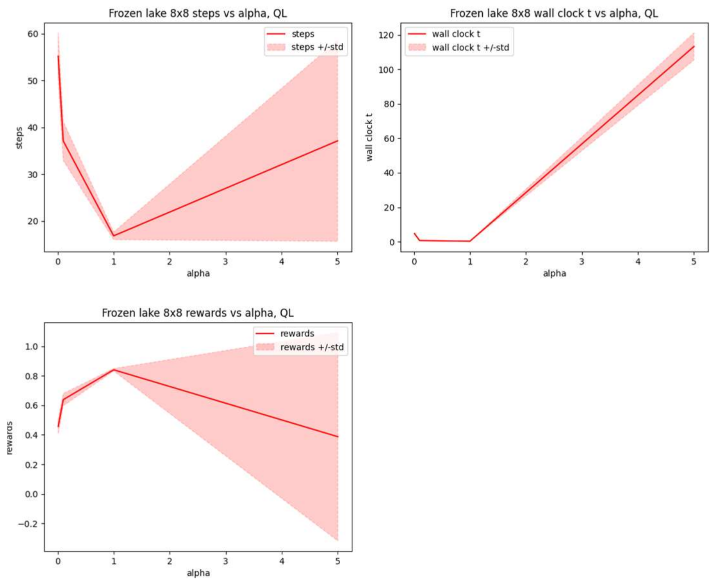


Figure 9 - Effect of α on QL. TopLeft: steps. TopRight: wall clock time. Bottom: rewards.

Another important hyperparameter is α, learning rate. In equation 4, α is used to determine how fast to update Q values. Similar to any typical machine learning algorithm, big learning rate may cause instability, and small learning rate may be too slow to learn. This same behaviour is exactly what we see here in Figure 9. When α is small as 0.01 and 0.1, the average reward at the end is quite low, and the number of steps is dramatically more. At α=1, we can see that the time it takes to train is the lowest, and the reward is the highest and consistent. The reward is not 1 because of the -0.01 reward shaping at every step the agent takes.

ε in QL determines exploration strategy during training. Coupled with greedy action, it forms ε-greedy action which balances between exploration and exploitation. At the beginning of the training, it is typical to have a large ε because the agent knows nothing of the world it is in, and large ε will allow the agent to take more random actions to get to new states which may be really good states to be in. As the agent learns, it is best to reduce ε so the agent takes greedy actions more than random actions. This is called ε decay. This is quite important in QL because if the decay is too fast, the agent likely only learns the good states to be in at the beginning of the game, which does not help to get to the goal. If the decay is too slow, the agent can still learn, but it will also have a higher chance taking random actions instead of the greedy ones, which also does not help the agent get to the goal.

Figure 10 shows the effect of ε decay. Initial ε is set to 1 for all cases, and minimum ε is 0.05. We can see that when ε is 0.99 amd below, the steps required are high and often the agent cannot reach goal, as shown by the big variance. When ε is at 0.9999 to 0.99999, there is some instability in the reward, likely due to the explained reason above that the agents still take more random actions. At ε=0.999, the steps are the lowest and the reward is the highest. This means at map size of 8, ε=0.999 is likely the best value. The wall clock time is low when ε is low, but it is not that meaningfull since the solution is nowhere close to optimal.

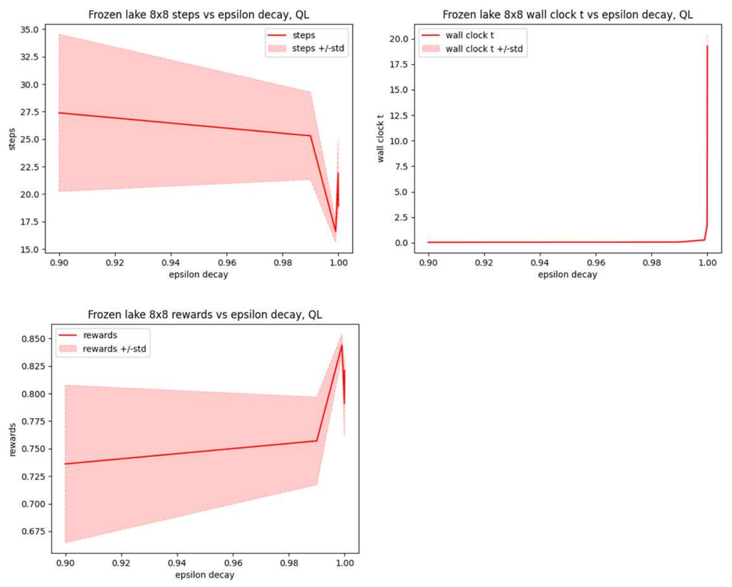


Figure 10 - Effect of ε decay on QL. TopLeft: steps. TopRight: wall clock time. Bottom: rewards.

The last hyperparameter to look at is the minimum ε. Ideally min ε should be higher than zero but still close to zero. The reason is a small positive value will allow the agent at the late stage of learning to occasionally explore to find a better state if needed while not veering off too much from the greedy actions. From

Figure 11, it is observed that when minimum ε increases as wall clock time required to converge. This is due to the excessive random actions that are not necessary at the late stage of training. It also decreases the chance of reaching the goal, which can be observed with the rewards plot on the top right.

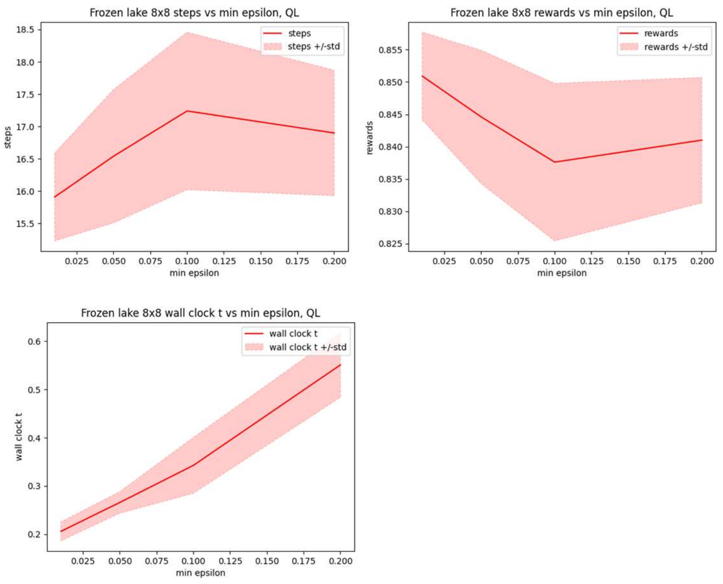
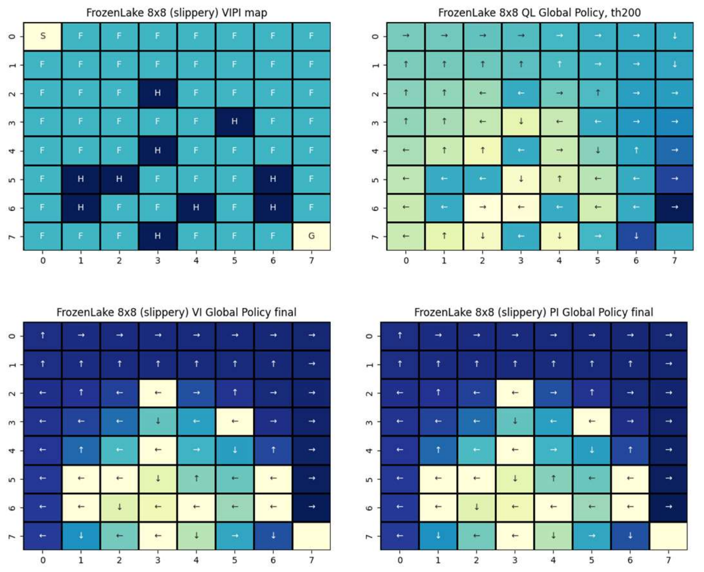


Figure 11 - Effect of minimum ε on QL. TopLeft: steps. TopRight: wall clock time. Bottom: rewards.

With the hyperparameters better tuned for frozen lake, we can again look at the stochastic action results among three algorithms. The gamma is set to 0.999 to maximize the chance to get to goal, even if it means it takes more steps to get there. From Figure 12, we can observe that VI and PI converge to the same global policy, which is not a surprise. They both result in leading the agent to go through the top towards right, and go down by pointing all arrow to the right to minimize chance of stepping on holes. For QL, we can see the majority of the policies is the same, but some key states have different policies, located on the top right hand corner. Some of the states are not pointing the agent to the top or right, which increases some chance of agent stepping on the holes due to stochasticity in actions. There are also some states on the lower left of the map that do not have matching policies between VI and QL. The reason QL does not converge to the same solution as VI or PI is because it does not have a defined state transition function, so it relies on sampling to estimate the state transition function. In addition, some of the states are visited less frequently because they are surrounded by holes, such as states in low-centre location of the map. If they are visited less, their estimated utilities will be less accurate and therefore the policies will be sub-optimal. Table 1 summarizes the training time and scores (10k runs, each successful run counts 1) of all three algorithms. VI is the fastest and it has a perfect score, which is the same as PI's score. QL does quite well while not a perfect score, while the training time is quite long. The "converge: 10 or 200" means the convergence criterion is set to 10 or 200 consecutive successful runs to goal. Note VI and PI have the advantage that state transitions are known. For QL to converge to the global optimal solution, it will need infinite number of samples to get the exact state transition function. While having finite samples, it does reach near optimal solution. Another point to make here is that the QL algorithm for frozen lake includes a reward of -0.01 for every step the agent makes that is not a terminal or absorbing state. This will change the agent's behaviour towards finishing the maze slightly faster, but it helps with QL training speed.



FrozenLake 8x8 (slippery) VIPI map
FrozenLake 8x8 QL Global Policy, th200
FrozenLake 8x8 (slippery) VI Global Policy final
FrozenLake 8x8 (slippery) PI Global Policy final

Figure 12 – Converged global policy with ϒ=0.999. TopLeft: map. TopRight: QL. BottomLeft: VI. BottomRight: PI.

Table 1 - Vi, PI, QL performance on probabilistic state transition frozen lake.

	Wall clock time (training)	Score
VI	1.41 sec	10000
PI	2.58 sec	10000
QL (converge:10)	10.15	9420
QL (converge:200)	115.43	9353

CART POLE

This RL problem has 4 continuous states, which were mentioned in the beginning. The default reward shaping is quite decent because for every iteration (steps) the pole does not swing over the limit, agent gets reward of 1. For the state transition functions, while not random since it follows deterministic classical physics, it is still unknown because we do not have the mass and size of the components, as well as the magnitude of the load applied by the actions. To run VI, we will need to discretize the states. If we have 10 bins per state, then we will have 10^4 states. Since we do not have state transition functions, we need to map all possible combinations which results in $(10^4)^2 = 10^8$ combinations. Lots of the state to state pairs won't occur because the system needs to follow rules of the physics, however the complete matrix is still required. The space complexity increases exponentially so having larger bins is not feasible.

The code for VI is from [12] with some custom modifications on number of bins and range of bins. VI in this case works by running many episodes, and in each episode, it counts the state to state transition occurrence for all steps, and then at the end of the episode, it uses the occurrence counter to update state transition probability, and then the state transition probability is used to run value iteration. Repeat the process until a set of episodes is reached. From Figure 13, we can see the training scores vs episodes for various bins per state (5, 6, 8, 10, 11, and 12). The plots are shown as 20 episode moving average. The raw scores are scattered and they are harder to read. The general trend is as number of bins increases, the training scores become higher at the end of 4000 episodes. Some important behaviours are for bins less or equal to 10, the training seems to hit some bottleneck and agent can only learn up to some average score. For 5 and 8 bins, it seems there are some breakthroughs in the second half of the training. This is likely contributed by randomness in the training, and the bins happen to capture important zones of the continuous states which can offer some advantage.

For 11 bins, it is hard to say if learning has hit a ceiling. For 12 bins, there is slow but steady learning throughout 4000 episodes. I'm unable to increase number of bins more than 12. At 12 bins, it occupies more than half of my 64GB RAM and by going to 13 bins, it will require $(13/12)^8 = 1.9$ times more RAM. Gamma for all runs here is 0.99, which is a reasonable number because $0.99^{500} = 0.00657$, which is not too small for the algorithm to ignore.

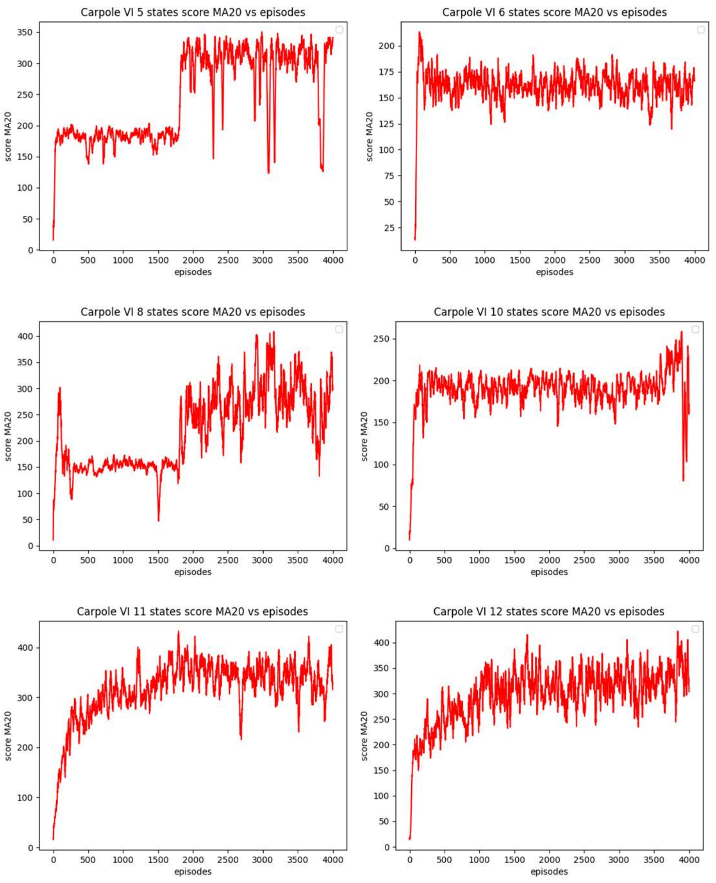


Figure 13 - VI training curves on discretized cartpole. "#" states in each chart title means number of bins per state observation. MA20 means 20-period moving average.

I chose not to extend number of episodes because it takes more than half day to train 12-bin configuration once, as shown in bottom right of Figure 14. Rest of the plots in the figure show the time in seconds it took vs episodes with various bins. Note that for 11 and 12 bins they seem to have 2 straight lines where at the beginning the slopes are steeper and they transition to shallow slope. This is due to the agent quickly fails an episode by not knowing what to do in early learning stage, so each episode is very short. It also learns quickly in early stage so the rate of increase in steps (also scores) of an episode goes up faster. The agent slows down learning after about 1000 episodes so the wall clock time also increases slower.

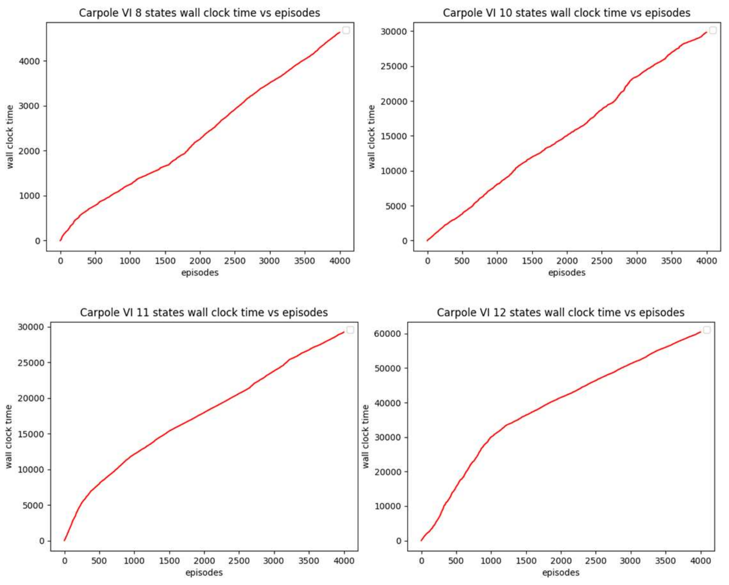


Figure 14 - VI training wall clock time on discretized cartpole. "#" states in each chart title means number of bins per state observation. MA20 means 20-period moving average.

Based on experiments done in this paper, PI will very likely behave similarly to VI. The limitation is the discretization of states which leads to astronomical amount of RAM required. Hyperparameter tuning is not feasible due to the time required to train. It is concluded that VI and PI can be done theoretically, but are not suitable in continuous state world. Instead, we will focus on Q-learning extension, DQN.

The code to run DQN comes from pytorch tutorial [9], with some customization done to include neural network hidden layer sizes as hyperparameters. The maximum number of episodes to train is set to 2000, and the convergence criterion is the agent can successfully keep the pole upright for 20 consecutive episodes. Besides the same hyperparameters as standard Q-learning, there are additional ones such as batch size and NN hidden layer size, which are explicitly required by neural networks. There are also memory replay size and τ, which are new and explained in the previous section. We will focus on the old hyperparameters first and move on to the new ones to understand their effects.

The effect of gamma on cart pole problem is shown in Figure 15. Similar to Q-learning in frozen lake, if gamma is too small, it becomes to short sighted and not able to account for rewards 500 steps out in the future. If gamma is too big, 500th step would have more impact on the Q values estimated by the neural network, implying more episodes and longer time required. From the top right plot, we can see between gamma=0.97 and 0.99, the trained model can hold the pole for 500 steps consistently.

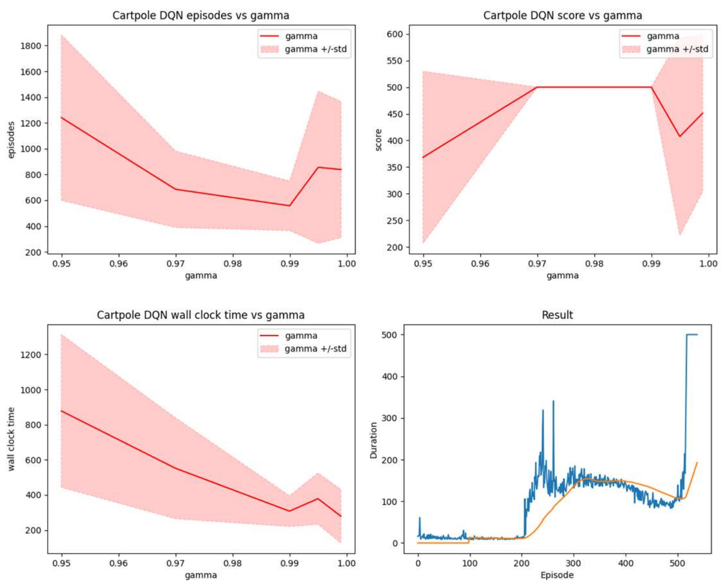


Figure 15 - DQN effect of ϒ on performance. TopLeft: episodes. TopRight: scores. BottomLeft: wall clock time. BottomRight: training curve at ϒ =0.99.

We can also see that episodes required to converge are higher outside gamma=0.97 and 0.99 range. It seems gamma=0.99 is the best in terms of training time and overall score. Bottom right shows a typical training curve of gamma=0.99 where y-axis, duration is the same as score per episode, and x-axis is number of episodes. Orange curve is the 100-episode moving average. The learning curve shows little improvement in the first 200 episodes. This waiting period can vary since there are other hyperparameters to tune, and also majority of the actions taken are comprised of random ones. When enough exploration is done, a series of good actions will prolong the pole in upright position. DQN will be able to learn from it so the scores shoots up right away.

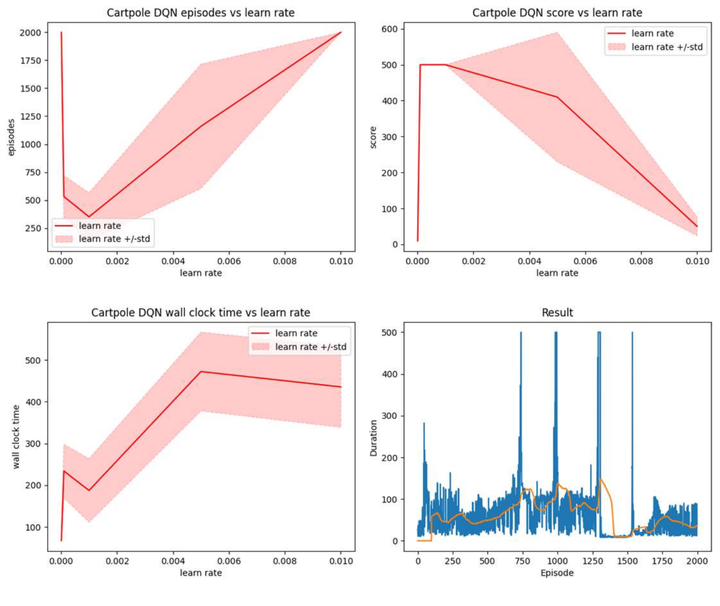


Figure 16 - DQN effect of learn rate on performance. TopLeft: episodes. TopRight: scores. BottomLeft: wall clock time. BottomRight: training curve at α =0.01.

The effect of learning rate is summarized in Figure 16. For DQN it is the same learning rate used on neural networks, which estimates Q values. It serves the same purpose of learning rate in standard Q-learning. We can see in the plots that cart pole is very sensitive to α, similar to standard Q-learning. α at 0.0001 to 0.001 are doing great and the wall clock time is also faster. Bottom right plot displays the training curve for α=0.01. When α is too large, the training becomes unstable and therefore not able to converge.

The ε decay is implemented differently from standard Q-learning although they serve the same purpose. In this case ε decay is represented by number of steps instead of a fraction. Concretely the decay is $e^{-(elapsed\ steps\ /\ \varepsilon\ \text{decay})}$[9]. In Figure 17 we can see at ε decay of 1000, it reliably finishes 500 steps comparing to others. This indicates values higher or lower may not provide a balanced exploration and exploitation strategy for cartpole problem.

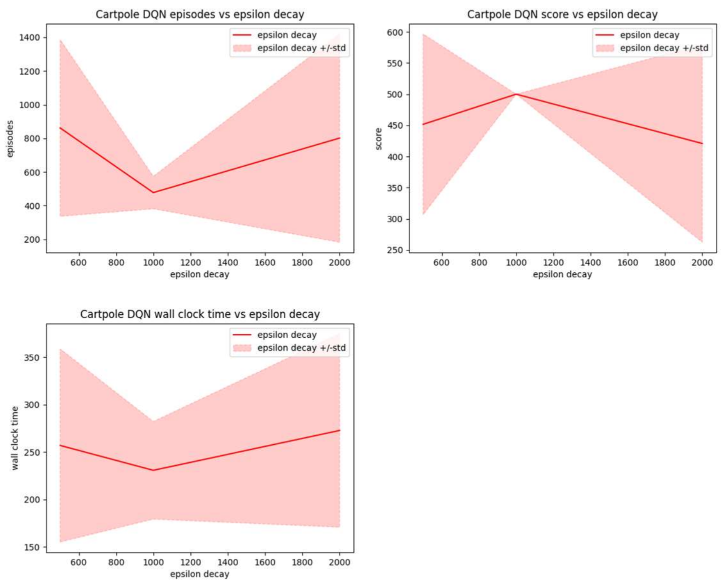


Figure 17 - DQN effect of ε decay on performance. TopLeft: episodes. TopRight: scores. BottomLeft: wall clock time.

Minimum ε describes how much exploration will take place when ε decay has converged to 0. An interesting observation here in Figure 18 is that min ε of 0.1 (10% of the time the agent will take a random action) results in good scores consistently. From my experience with other RL problems, min ε of 0.1 is on the high side. This is likely due to the type of the problem we face here. In frozen lake, there is considerably higher probability of taking a random action and land in a hole right away and ends the episode. In cartpole, when the pole is steadily upright, and a random action will not immediately swing the pole outside allowed angles due to momentum of the system. Min ε higher than 0.1 poses more risks of not reaching upright pole in the first place so the average score reduces significantly.

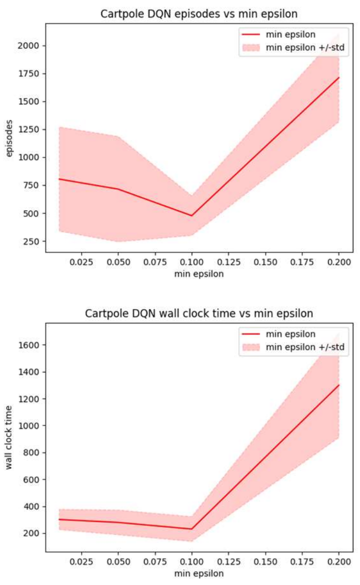


Figure 18 - DQN effect of minimum ε on performance. TopLeft: episodes. TopRight: scores. BottomLeft: wall clock time.

Batch size is a hyperparameter used in neural networks. In supervised learning setting, typically if batch size is too small, learning curve is noisier due to low statistical representation of the training data. If the batch size is too big, it is more difficult to do gradient descent, which leads to slower learning. It is hard to tell if learning curves behave the same in RL because the y labels are moving targets. Instead we can see how it affects the speed and scores. In Figure 19, we see that when batch size increases, it helps with training speed and scores as well. The reason is the weights are updated once per step, and there is also a soft update (will be discussed later) that limits how fast the weight updates. These stabilize learning in DQN and as a result DQN trains faster.

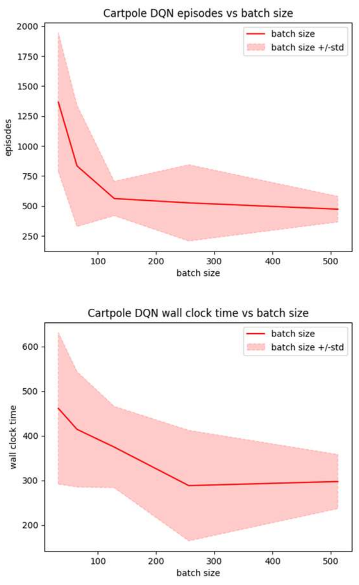


Figure 19 - DQN effect of batch size on performance. TopLeft: episodes. TopRight: scores. BottomLeft: wall clock time.

Neural networks in DQN estimate Q functions. Bigger NN can estimate more complicated Q functions. Bigger NN could lead to overfitting so it's important to study this hyperparameter. In this DQN setting, there are two hidden layers. Only layer 1 analysis is presented, as shown in Figure 20. While layer 2 hidden layer size is fixed to 128, hidden layer 1 size shows that the neural networks are not able to learn well below size of 128. With size of 256, it is able to score 500 consistently, and the training time is also faster than size of 128 on average. This is likely due to the use of GPU which can parallelize computations, as well as better fitting capability from higher dimensional inputs. Overfitting is minimized by setting criterion that stops training when 20 consecutive episodes score 500.

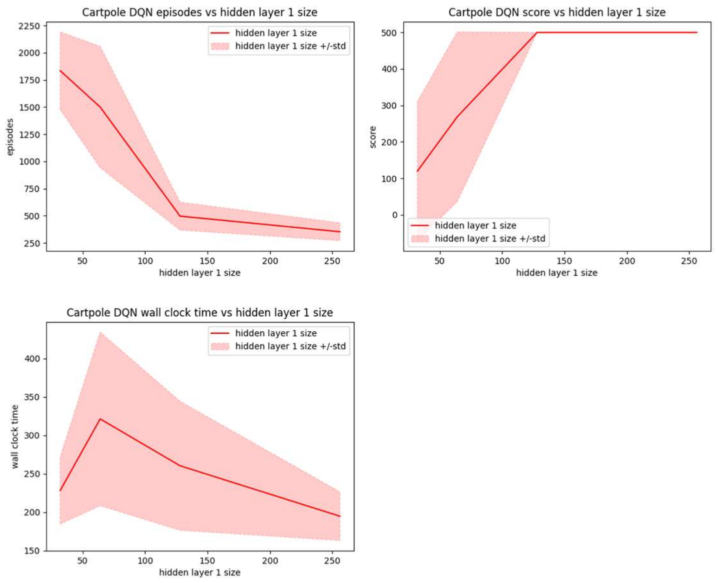


Figure 20 - DQN effect of hidden layer 1 size on performance. TopLeft: episodes. TopRight: scores. BottomLeft: wall clock time.

Memory replay size is to store state, action, next state, reward tuples for neural networks to learn. Intuitively, bigger memory replay should help NN learn better because it increases the training data size. The plots in Figure 21 tell a different story. For memory size 10k and below, trained scores are perfect and the episodes to complete training are quite reasonable. At size of 15k, not all trainings went well, shown on the top right plot. It is possible that the failure occurred by random. However, the observed behaviour is that larger memory size will include older episodes of the training which are not very good samples for NN to learn. When these samples are drawn by random, they steer the neural network weights in the wrong directions. We can see the effect in the training curve shown in bottom right, which has a memory size of 20k. In this instance the training did meet the convergence criterion, but we can also see the agent was able to complete 500 steps before 350 episodes, but it struggled a while before getting 20 consecutive episodes. When memory size is 10k or smaller, this behaviour was not present.

The last hyperparameter is τ, used to regulate how much influence we want from the target network weights to the policy network weights, which is used to determine greedy actions. The formula is shown below [10]:

$$\varphi \leftarrow (1 - \tau)\varphi + \tau\theta \qquad (5)$$

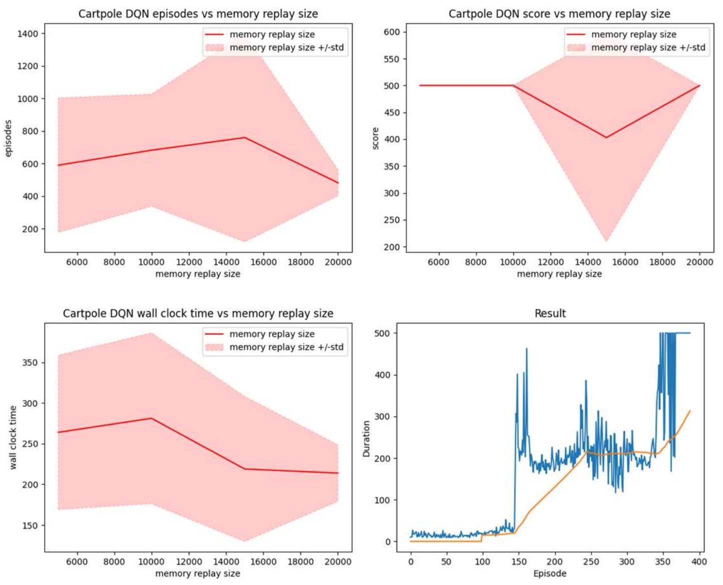


Figure 21 - DQN effect of memory replay size on performance. TopLeft: episodes. TopRight: scores. BottomLeft: wall clock time. BottomRight: training curve at memory size=20k.

φ is the policy network, and θ is the target network. This is one of the key implementation to stabilize DQN training. If τ is too big, training is unstable. If τ is too small, NN won't be able to learn in time. The plots in Figure 22 agree with the statement. We can see that between 0.003 and 0.005, the scores are consistently 500 while bigger or smaller τ result in lower average score and longer training episodes. One interesting behaviour is that only low τ results in longer wall clock time. This is because when training is slow due to low τ, the number of steps per episodes are still as high as a few hundreds during training. When τ is high, the training is unstable so the number of steps can be a lot lower which makes the training wall clock time faster.

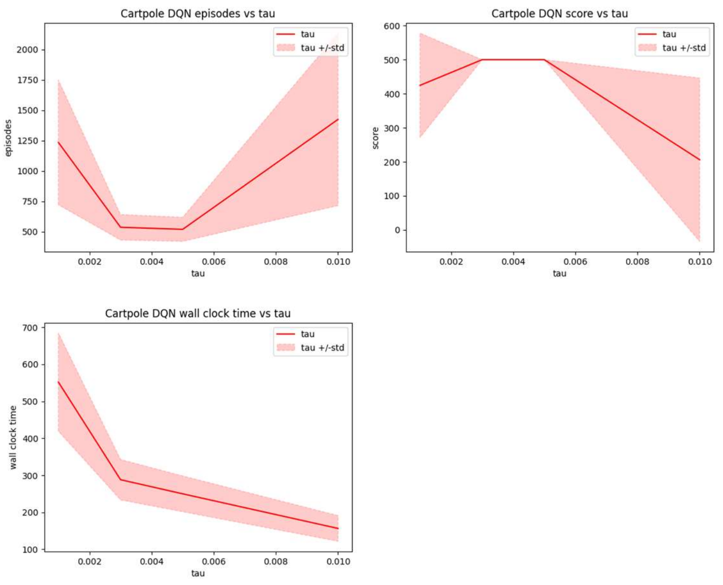


Figure 22 - DQN effect of τ on performance. TopLeft: episodes. TopRight: scores. BottomLeft: wall clock time.

Table 2 summarizes the tuned hyperparameters of the DQN. It was trained 10 times and all of them were able to converge with average required episodes of 291.6. A test run shows that the pole stays perfectly straight while DQN also learned to keep the cart right in the centre with anything barely moving, shown in Figure 23. It is quite amazing DQN learned the implicit rules of physics to balance the pole.

Table 2 - Tuned DQN hyperparameters on cartpole problem.

BATCH_SIZE:256	GAMMA:0.99	EPS_START:0.9	LR:1e-3
EPS_DECAY:1500	TAU:0.005	EPS_END:0.05	L1:256
MEMORY:13000	CONVERGE_CRITERIA:20	EPISODES:2000	L2:128

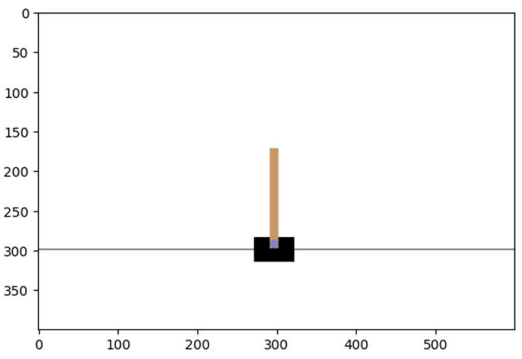


Figure 23 - Last frame (500th) of a cartpole episode by tuned DQN.

VII. CONCLUSION

In this paper we analyzed VI, PI, and QL (DQN) on two different RL problems. VI and PI are great for discrete states with known state transitions while QL (DQN) can solve discrete state problems with unknown state transitions, it is more robust on continuous state problems.

VIII. REFERENCES

1. https://en.wikipedia.org/wiki/Markov_decision_process
2. https://en.wikipedia.org/wiki/Markov_property
3. https://www.gymlibrary.dev/environments/toy_text/frozen_lake/
4. https://www.gymlibrary.dev/environments/classic_control/cart_pole/
5. https://en.wikipedia.org/wiki/Bellman_equation
6. https://en.wikipedia.org/wiki/Dynamic_programming
7. https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node20.html
8. RL book
9. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
10. https://arxiv.org/pdf/2008.10861.pdf
11. https://github.com/jlm429/bettermdptools
12. https://gist.github.com/kevinstumpf/8c6f7af4c94bbaa8cacc859891c090dd