Project Report
Q-Learning, DynaQ and DeepQ Implementation on FrozenLake 8x8

CS4100: Artificial Intelligence

Anni Yuan, Fengyi Quan
Northeastern University

The problem

Agents are learning from rewards and punishment to achieve a better performance, like humans, we also learn from making mistakes. Therefore, we have decided to focus on reinforcement learning. Our goal for this project is to apply Q-Learning and take one step further to implement simple Deep Q-learning and Dyna Q-Learning to play an Toy text game - FrozenLake8x8 using OpenAI framework. The goal of this game is to find a safe path across a grid of ice and water tiles that is to go from tile S to tile G without falling into title H.

State Space:	State = tile, ranged from [0,63] inclusive, therefor state space is 64
Action Space:	Up, down, left, right(4 in total)
Reward:	In-game reward function is used. The agent is rewarded 1 point if reaching tile G, and 0 otherwise.
Initial State:	State = 0 (Represents tile S, located in the upper left corner)
Successor Function:	We used the the step function to the the next state next_state, reward, done, info = env.step(action)
Terminate	If done == True, then the game ends.

Algorithms

Q-Learning

Q-learning is a model-free value-based reinforcement-learning algorithm that updates Q-table based on each action in a particular state. It uses previous experience to predict approximation of Q-value, transition state, reward.

Dyna Q-Learning(DynaQ)

Dyna Q-learning maintains the process of Q-learning and adds a planning component. The learning and planning processes are almost the same. However, the data sources are different. Learning is from real experience, and planning is from simulated experience based on the model with the assumption of determinism. Model contains state, action, next state and reward tuples. And it can be improved in the planning part. So in each action taking, the learning process is strengthened by updating the Q function from both actual action taking and model simulation.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Deep Q-Learning(DeepQ)

Deep Q-Learning (DQN) is an extension of Q-Learning. It uses a convolutional neural network to replace Q-table resulting in a more stable learning process. Since Q-learning has a bad performance when states are very large, DQN uses a neural network to estimate the optimal Q-function instead of calculating Q-value directly. The network receives the state as an input and outputs the Q values for all possible actions. All the past experience is stored in the memory.

The main difference between Q-learning and DeepQ is that DeepQ can handle environments that involve continuous action and states. Since DeepQ has a replay buffer to store the consecutive actions and states, it can break the correlation between consecutive samples. In a continuous environment, it is likely that consecutive states and actions have some correlation.

Our implementation is based on a DeepQ implementation on FrozenLake 4x4 available at link1 [1]

[1]https://github.com/Amber0914/Reinforcement_Learning_Algorithms/blob/master/Q-Learning/deep_frozenlake_unslippery.py.

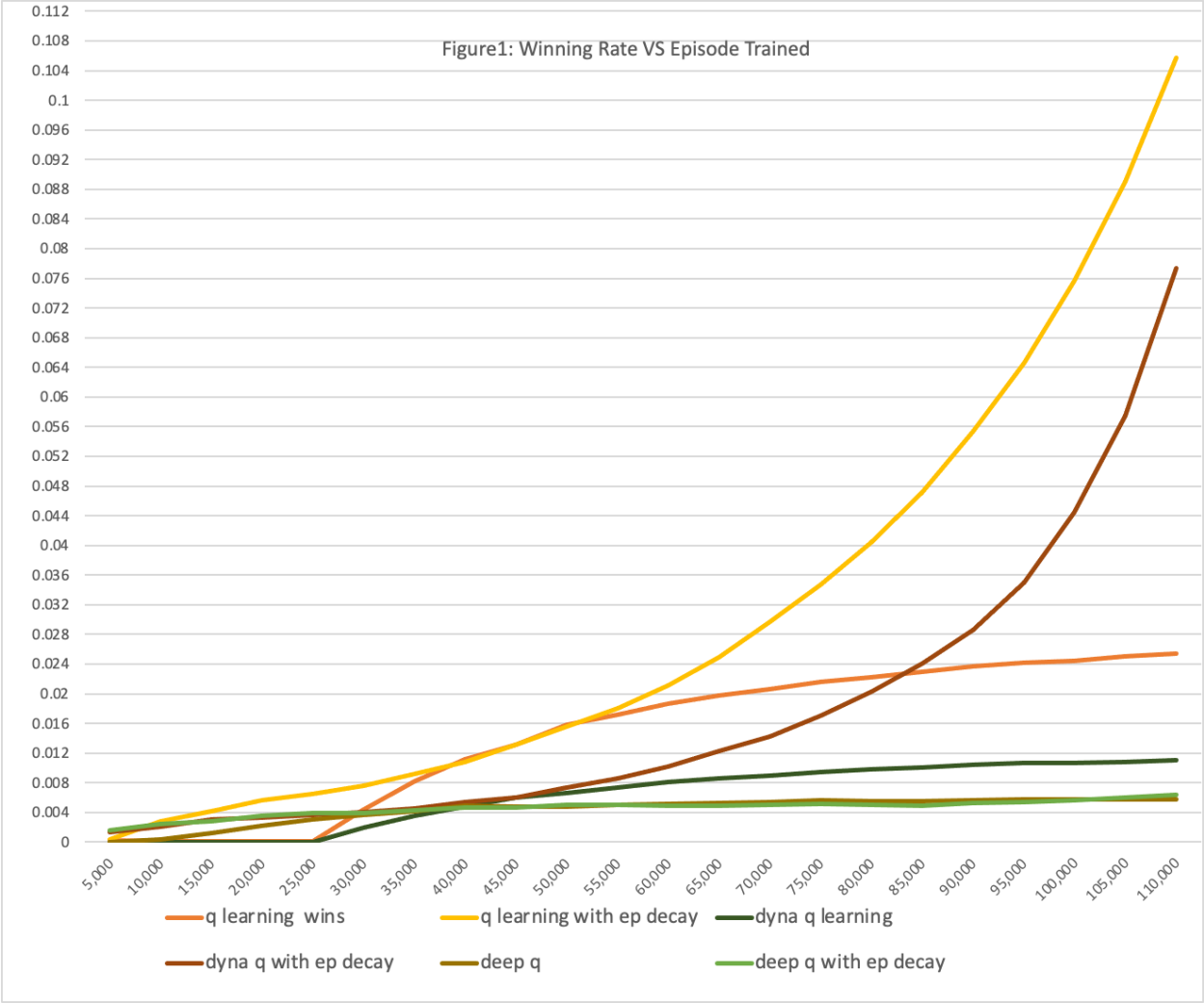
```
Initialize Q_0(s,a) for all pairs (s,a)
s = initial state
k = 0
while(convergence is not achieved)
{
    simulate action a and reach state s'
    if(s' is a terminal state)
    {
        target = R(s,a,s')
    }
    else
    {
        target = R(s,a,s') + γ max_a' Q_k(s',a')
    }
    θ_{k+1} = θ_k − α Δ_θ E_{s' P(s'|s,a)} [(Q_θ(s,a) − target(s'))^2] |_{θ=θ_k}
    s = s'
}
```

Methodology and Results

Three models were trained twice using Q-Learning, DynaQ with steps of 5 and DeepQ. Each model is trained 11,000 episodes, winning rates are documented every 5000 episodes. Below are the parameters for each round.

1. learning rate = 0.1, epsilon = 0.5, gamma = 0.9, max_step = 100
2. learning rate = 0.1, epsilon = decays linearly from 0.9 to 0.001, decay = 1/Episode Trained ,gamma = 0.9, max_step = 100

Episode Trained	(epsilon = 0.9,learning rate = 0.1,gamma = 0.5, max_step = 100, steps = 5, decay = 1/Episode Trained)					
	q learning	q learning with ep decay	dyna q learning	dyna q with ep decay	deep q	deep q with ep decay
5,000	0.0002	0.0004	0	0.0014	0	0.0016
10,000	0.0001	0.0028	0	0.0021	0.0004	0.0024
15,000	0.000133333	0.00413	0	0.003	0.001266667	0.0028
20,000	0.0001	0.0056	0	0.00325	0.00215	0.0035
25,000	0.00008	0.00648	0	0.00364	0.00308	0.00392
30,000	0.0044	0.0076	0.0019	0.00403	0.003733333	0.00393
35,000	0.008257143	0.0092	0.003571429	0.00457	0.004142857	0.00431
40,000	0.0112	0.01083	0.004725	0.00538	0.004925	0.0047
45,000	0.013133333	0.01309	0.006	0.00604	0.0048	0.00469
50,000	0.0158	0.01556	0.00666	0.0074	0.00476	0.00498
55,000	0.017163636	0.01809	0.007345455	0.00853	0.005036364	0.00498
60,000	0.0187	0.02117	0.008066667	0.01015	0.0052	0.00487
65,000	0.019815385	0.02497	0.008553846	0.01229	0.005292308	0.00491
70,000	0.020671429	0.02969	0.008942857	0.01421	0.005342857	0.00503
75,000	0.02164	0.03469	0.009453333	0.01709	0.0056	0.00511
80,000	0.0222	0.04034	0.0097875	0.02024	0.0055125	0.00499
85,000	0.023	0.0472	0.010105882	0.02406	0.005494118	0.00495
90,000	0.023711111	0.05539	0.010377778	0.02866	0.0057	0.00521
95,000	0.024136842	0.06465	0.010621053	0.03504	0.005747368	0.00543
100,000	0.02439	0.07569	0.0107	0.0444	0.00579	0.00563
105,000	0.025	0.089	0.010809524	0.0575	0.005761905	0.00597
110,000	0.025472727	0.10579	0.010990909	0.07731	0.005809091	0.00634



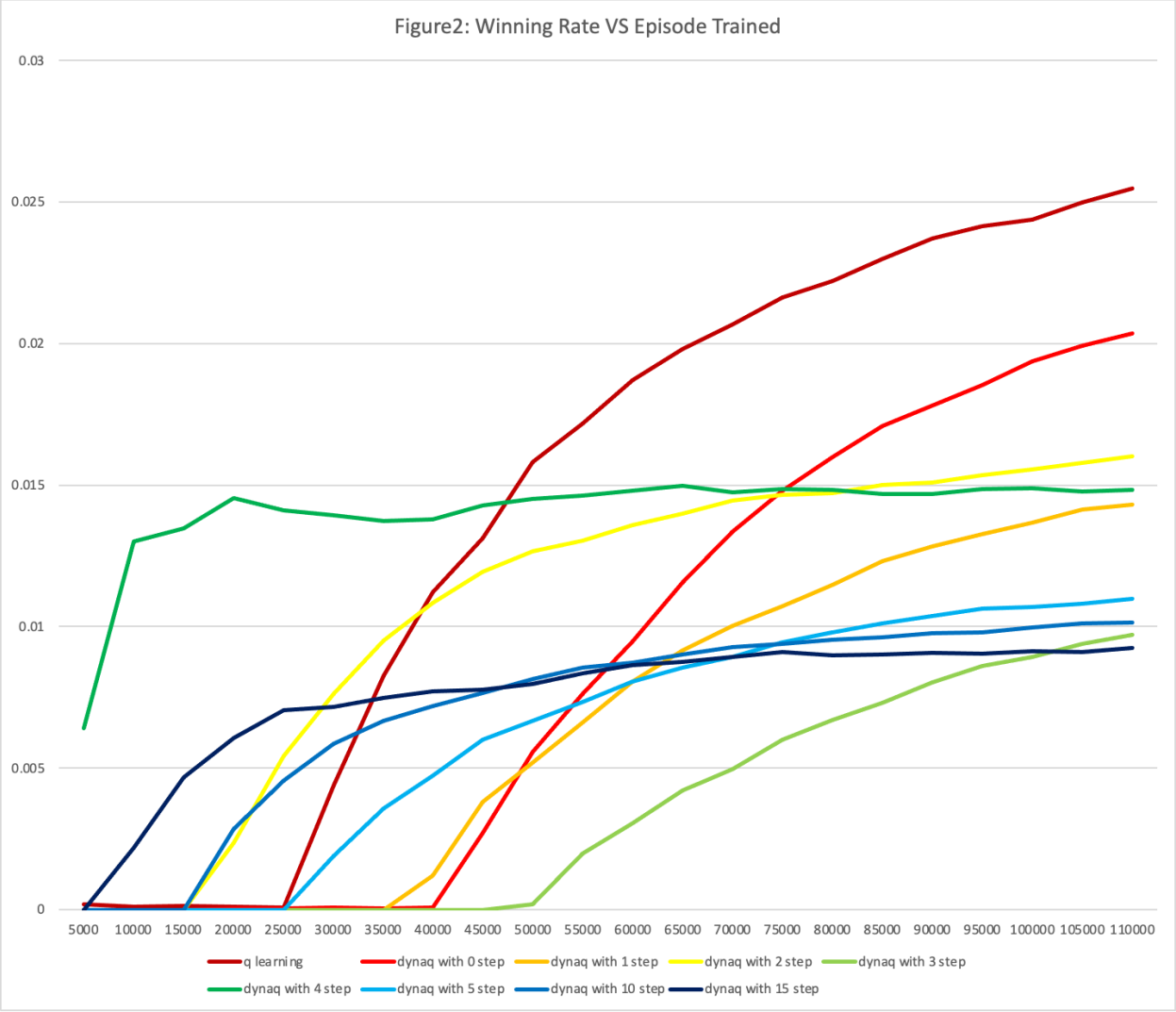
Improvements/Variation

From the above data, Q-Learning with epsilon decay and DynaQ with epsilon decay performs significantly better than the others. Both of them grow exponentially, while DeepQ with and without epsilon decay almost show no learning. It can also be seen from Figure1, algorithms with epsilon decay outperform the algorithms without epsilon decay. This is because we do not have any information about the model at the early stage, and thus we want to explore new states to gain experience which requires higher epsilon. As we have more experience, it then makes more sense to learn from the experience itself by following the policy which requires smaller epsilon.

In addition, because Q-Learning gives an optimal action when dealing with a finite Markov Decision Process and our state space is relatively small. It is expected that Q-Learning will have great performance. However, DynaQ and DeepQ do not meet our expectations, we expected DeepQ to have the best performance and DynaQ the

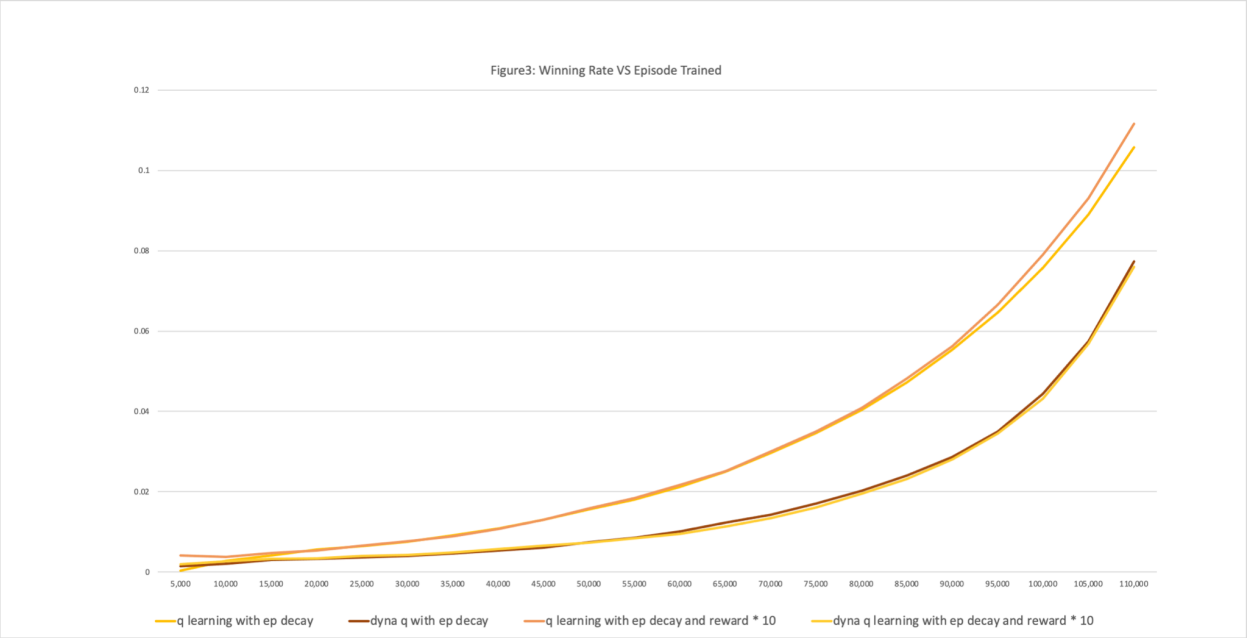
second best. Therefore, we decided to modify the algorithms further to improve the performance. To do this, scaling rewards are applied to Q-Learning with epsilon decay and DynaQ with epsilon decay, prioritized experience replay is applied to DeepQ with epsilon decay.

As the performance of DynaQ is unexpected and there are multiple factors that could impact the performance such as bad implementation or bad choice of the planning step. DynaQ with different planning steps are run to verify the implementation and to investigate the impact of the planning steps. Results along with the Q-Learning results are shown in Figure2. It can be verified that Q-Learning and DynaQ with zero planning step have the same learning rate as the curves are kind of parallel to each other. Compared to Q-Learning, DynaQ generally learns faster, meaning it can achieve the same winning rates with less number of trained episodes. However, the below figure also shows DynaQ stabilizes at a lower winning rate when the planning step increases.



Scaling Rewards

Scaling rewards performance well when our future rewards have a large range (eg: future rewards scattered from 1 to 10,000). By scaling the rewards, it helps to learn from a more balanced distribution of rewards. Since our reward function gives 1 to the goal tile and 0 otherwise. We plan to give the goal state a higher weight and see if it can improve our learning algorithm. By scaling rewards by 10, we deem that it will favor the goal tile more than before. Below is the result by scaling the reward by 10.



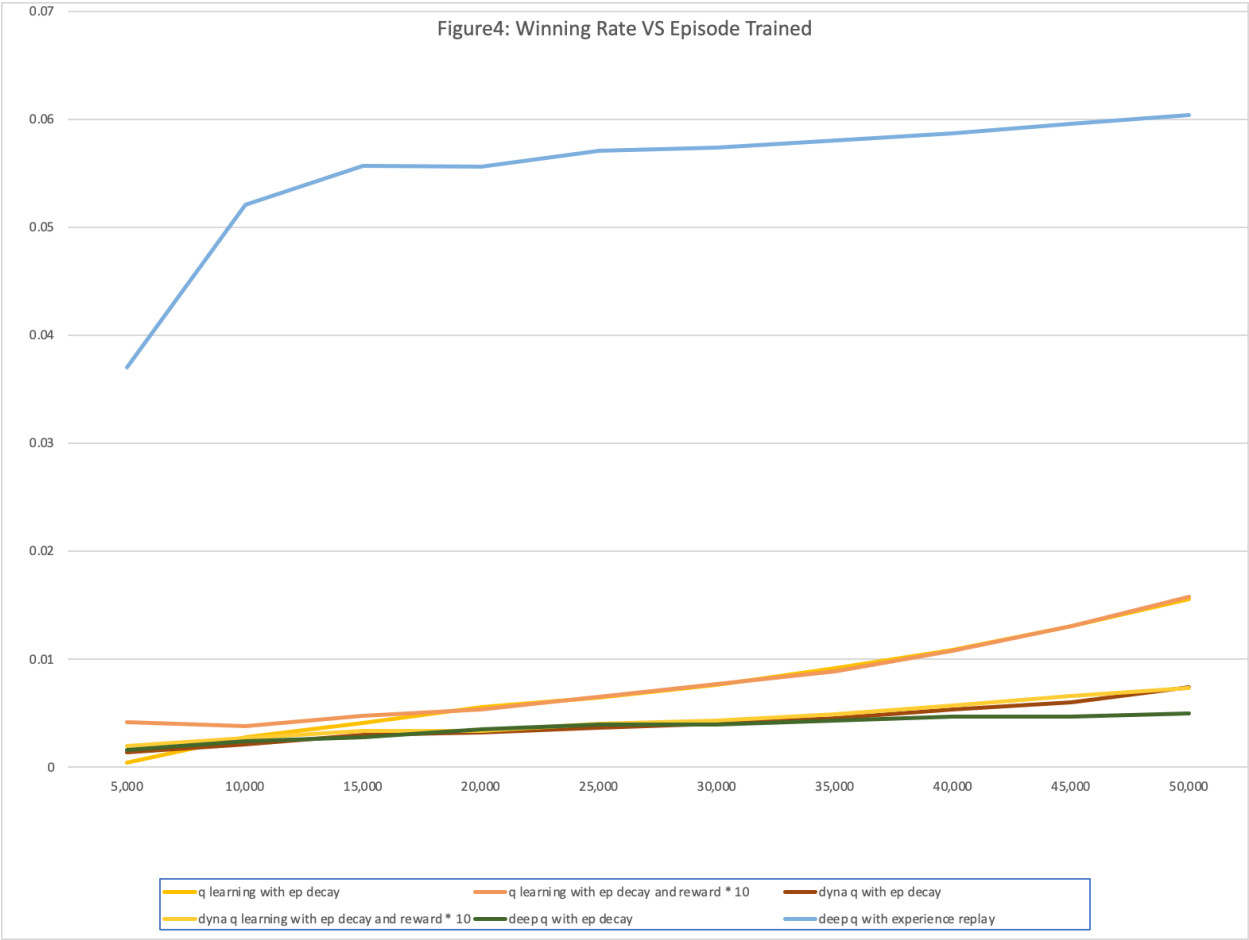
Prioritized Experience Replay(PER)

Experience replay is a replay memory technique used in reinforcement learning. There is a buffer/memory that stores a large number of agent's experiences at each time step. In the process of training, a minibach of experiences is sampled randomly and replayed after each episode.

Different from experience replay, prioritized experience replay does not sample the minibatch randomly, it plays more important experience more frequently. The importance of each experience is measured by the temporal-difference error.

Our implementation of prioritized experience replay and DQN is based on the DQN with prioritized experience replay implementation on Cartpole-V0 available at link2[2].

[2] https://github.com/jcborges/dqn-per



Discussion

From Figure3, it is found that there is no clear difference by applying scaling rewards. We then realize the reason behind it might be the trends and the behavior corresponding with the highest q-value are the same because we do not have immediate rewards except at the end of the games. Therefore, it is reasonable that there is no effect on the agent learning process.

From Figure 4, we can see DeepQ with PER perform the best which is expected. Even though the learning rate is kind of linear after 15,000 episodes, it does achieve a relatively high winning rate at a small number of training episodes, and has a good learning speed until 10,000 episodes are trained. Overall, it can also be interpreted from Figure 4 that DeepQ with PER will stabilize at a higher winning rate than other algorithms.

Our first DeepQ used Tensorflow but our second DeepQ used Keras. In terms of the implementations, we found Keras is more user-friendly as it can achieve the same result

with fewer lines of codes. In terms of speed, even though we have applied experience replay and it is prioritized, we do feel Keras takes more time than Tensorflow (Keras takes three days to train 50,000 episodes while tensorflow only takes a few minutes).

We used Mac Pro for this project and one interesting thing we have noticed is that if our laptops are working on intensive tasks like training 11,000 episodes non-stop, then the results would not be as good as if we make the algorithms sleep for five seconds after every 5,000 episodes.

To conclude, DeepQ with PER works the best for FrozenLake8x8, but it is extremely time consuming. DeepQ without PER is relatively quick, but the learning results are undesirable. In general, DynaQ learns faster than Q-Learning but if we train the model with enough episodes, Q-Learning actually works better. The reason why DeepQ without PER has worse performance than Q-Learning might be that DeepQ has a much more complex hidden network. It would cost a huge amount of time to build these hidden layers. This could take numerous episodes to learn. So we could not have the desired result at the beginning. Since we only have 11,000 episodes, it might not be enough to make an accurate network. If we increase our training episodes, we might have a network to give us more optimal actions in each step.

Future Directions

One possible direction is to make agents differentiate frozen tile and holes more clearly. Since it gives 0 reward for both frozen tile and holes, the agent would not reflect this difference obviously only when it reaches the goal tile. There is no penalty if the agent falls into the hole, thus, it will never learn how to prevent it. The possible improvement can be given -1 rewards to the agent if it falls into the hole. In the current scenario, we found it can learn very fast if we give an agent a map without any holes in it. But it cannot even reach the goal if there are multiple holes in it. If we give a penalty when it falls. This would improve a lot.

In this project, we used ‘relu’ for hidden layers and ‘linear’ for the output layer to build the network. ReLU is the most common function used for a hidden layer, it is used to calculate the maximum. It is faster than ‘sigmoid’ and ‘tanh’. If time permits, we might use different activation functions to see how it affects the agent learning process

After understanding the basic concept of DeepQ learning algorithm and applying it to a simple text game, one future direction is to take a step further and apply DeepQ learning algorithm to image based games. Because it is hard to tell the game status

from just one frame, according to a tutorial available on Link3[3], game states are now a deque of four frames. By observing the prior sequence of frames, we will know if the target is moving to the right or left and thus choose an action.

References

Zhang, J., 2019. *Reinforcement Learning—Model Based Planning Methods*. [online] Medium. Available at: <https://towardsdatascience.com/reinforcement-learning-model-based-planning-methods-5e99cae0abb8> [Accessed 28 April 2021].

Gautam, A., 2018. *Introduction to Reinforcement Learning (Coding Q-Learning) — Part 3*. [online] Medium. Available at: <https://medium.com/swlh/introduction-to-reinforcement-learning-coding-q-learning-part-3-9778366a41c0> [Accessed 28 April 2021].

GeeksforGeeks. 2019. *Deep Q-Learning - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/deep-q-learning/> [Accessed 28 April 2021].

Medium. 2019. *Zero to One: (Deep) Q-learning, Part1, basic introduction and implementation*. [online] Available at: <https://medium.com/@qempsil0914/zero-to-one-deep-q-learning-part1-basic-introduction-and-implementation-bb7602b55a2c> [Accessed 28 April 2021].

Paperswithcode.com. 2021. *Papers with Code - Experience Replay Explained*. [online] Available at: <https://paperswithcode.com/method/experience-replay> [Accessed 28 April 2021].

Paperswithcode.com. 2021. *Papers with Code - Prioritized Experience Replay Explained*. [online] Available at: <https://paperswithcode.com/method/prioritized-experience-replay> [Accessed 28 April 2021].

Danieltakeshi.github.io. 2016. *Frame Skipping and Pre-Processing for Deep Q-Networks on Atari 2600 Games*. [online] Available at: <https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/> [Accessed 28 April 2021].

[3] https://github.com/simoninithomas/Deep_reinforcement_learning_Course/blob/master/Deep%20Q%20Learning/Space%20Invaders/DQN%20Atari%20Space%20Invaders.ipynb