Project Report
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 QLearning and take one step further to implement simple Deep Q-learning and Dyna-Q 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, -1 if falling into tile H, 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

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

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 deep Q-learning is that deep Q-learning can handle environments that involve continuous action and states. Since deep Q-learning 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 Deep Q Learning implementation on FrozenLake 4x4 available at 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, Dyna Q-Learning with steps of 5 and Deep learning. Each model is trained 11,000 episodes, results are documented every 5000 episodes. Below are the parameters for each round.

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

Episode Trained	Average score / Win					
	First Round: epsilon = 0.9			Second Round:epsilon -= 1/Episode Trained		
	Q-Learning	Dyna Q-Learning (step = 5)	DeepQ	Q-Learning	Dyna Q-Learning (step = 5)	DeepQ
5,000						
10,000						
15,000						
20,000						
25,000						
30,000						
35,000						

40,000						
45,000						
50,000						
55,000						
60,000						
65,000						
70,000						
75,000						
80,000						
85,000						
90,000						
95,000						
100,000						
105,000						
110,000						

Improvements/Variation

From above data, DynaQ shows significant improvement while Deep Q shows a slightly worse performance. Therefore, we decided to further improve these two algorithms.To do this, scaling rewards and prioritized experience replay are applied to DynaQ and Deep Q Learning separately.

Scaling Rewards

Since the future reward may be extremely large or small, it will have a different effect on the learning process based on how the reward function is defined. Since our reward function gives 1 to the goal tile and 0 otherwise. We plan to scale rewards to have a more balanced learning process and to see if it can improve our learning algorithm. This does not affect the functionality of the value function, because the behaviors and trends of the action with the highest reward value would be the same.

By scaling rewards by 10, it will most likely go toward the goal tile. To do this, we think the learning speed would be pretty slow and the training episode might not be that large. We try to increase the reward to make the agent go towards the goal tile likely. Below is the result by increasing the reward by 10.

learning rate = 0.1, epsilon = 0.9,gamma = 0.9, max_step = 1000

Episode Trained	Winning Episodes / 5000 Episodes
	Dyna Q-Learning (step = 5) epsilon changing
5,000	2780
10,000	2876
15,000	2719
20,000	2868
25,000	2702
30,000	2748
35,000	2575
40,000	2639
45,000	2938
50,000	2675
55,000	2492
60,000	2805
65,000	2700
70,000	2809
75,000	1664
80,000	2664
85,000	2614
90,000	2694
95,000	2405
100,000	1855

On the contrary, by narrowing down the reward by 1/10, we can see the learning process is very slow.

Prioritized Experience Replay
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 usually 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.

Discussion
In terms of implementation of Deep Q learning, 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 experience replay and it is prioritized, we do feel Keras takes more time than Tensorflow (Keras takes one day to train 10,000 episodes while tensorflow only takes a few minutes). .....

Future Directions
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 algorithm to image based games. ……

References:

https://towardsdatascience.com/reinforcement-learning-model-based-planning-methods-5e99cae0abb8

https://medium.com/swlh/introduction-to-reinforcement-learning-coding-q-learning-part-3-9778366a41c0

https://www.geeksforgeeks.org/deep-q-learning/

https://medium.com/@qempsil0914/zero-to-one-deep-q-learning-part1-basic-introduction-and-implementation-bb7602b55a2c

https://paperswithcode.com/method/experience-replay

Milestone

Report:
For now we've implemented Q-learning, DynaQ and DeepQ learning for the toy text Game-FroxenLake8x8.The next stage for this project is to train each algorithm starting from 5000 episodes to 100,000 episodes (increment by 5000). After each train, we will play the game 5000 times, and document the number of wins.

Reflect:
Instead of a toy text game our original milestone is to finish implementing Q-Learning, DynaQ and DeepQ Learning and to apply them to an Atari game- SpaceInvaders. The reason we changed the game is because we underestimated the difficulty of implementing those three algorithms to an image based game. The observation of SpaceInvaders is an RGB image of the screen which is an array of shape (210, 160, 3). Even though we could convert each observation into a grayscale image of shape (110, 80) that leaves us a state space of 256^(110x80) which is too large to apply Q-Table, that means we cannot apply both Q-learning and DynaQ. We've also looked at the implementation of Deep Q learning to an image based game which is quite complicated. As our goal is to implement the three algorithms and have a fundamental understanding of how they work, we decided it is better to switch to a simpler game.

Replan:
After gathering all the data, we will start the draft report and analysis. The analysis will focus on which algorithm works the best. We may also compare the performance of DynaQ when its steps are 5 and 10.
We already have the data for some basic algorithms. The next thing to do is to try to optimize the algorithm by applying scaling rewards and possible methods.