GAME BOT: Automating game play using Reinforcement Learning

Ragavendran Balakrishnan

MCS, North Carolina State University
rbalakr2@ncsu.edu

ABSTRACT

This paper demonstrates the value that reinforcement learning (RL) brings to the Machine learning domain. For any agent built with an reinforcement algorithm,works on the concept that the actions that are taken in the environment, that maximizes the future cumulative rewards, ought to be the right set of actions. All the concepts about Reinforcement learning and the related strategies in it will be shown in the Section 2 of this paper. The paper then shows the implementation of RL using a Game Bot agent, in the Sections 3.We will be seeing one of the strategy of RL- Q learning algorithm for implementing the Game Bot along with the use of openai gym module in python. The results both metrically and visually will be showing the functioning of RL in section 4.

1. INTRODUCTION

Most of the games that we see in the real world are dynamic in nature that are played by human agents. The human agent(or simply humans) actually learn by series of trial and error actions and finally apply the experience gained in playing and reach their goal[1]. The randomness that we see in the gaming domain is what makes it difficult to learn and that requires an algorithm, which can learn by the experience gained given from an environment. This makes it a perfect candidate for demonstrating Reinforcement learning.

A Game bot is an automated program that plays the given game on behalf of a human player[1]. It is an Intelligent agent that works without any/less human intervention. Like any other intelligent agent, it requires a lot of data about the environment to model its policies/ rules for reaching its goal. The vastness of the gaming domain requires the environment(game) to be representable.

Over the past few decades, Reinforcement Learning (RL) has been successfully applied to a variety of field including the game theoretic decision processes and Robo Cup soccer . It has also been applied successfully for a number of computer gaming applications including real-time strategy games , backgammon, and more recently for first-person shooter (FPS) games[2].

Most of the games of today's generation comes with countess states with complex environments. These games requires so much computing power to be rendered and played properly. But, adding a layer of automation bot, means a lot of data has to be stored and retrieved for making the bot to learn. Since the primary focus of this paper is to explain the working of RL in Game Bot and not in developing a advanced game, we will be looking at a simple 1980's atari game, called the 'mountain-car' that has all the needs of a full fledged RL and also not so complex that it needs a lot of computational power to implement.

In this paper, I will also be focusing on the various strategies of RL that serves the need for devising our game bot and finally, I will be presenting the details of the implementation and results of one of the strategy that I used for implementing my game bot.

2. REINFORCEMENT LEARNING

The goal of reinforcement learning (Sutton and Barto, 1998) is to learn good policies for sequential decision problems, by optimizing a cumulative future reward signal[3]. The notion of cumulative reward implies the total rewards earned by an agent in reaching the current state.

A typical RL setup comprises of mainly two components : An agent and the environment. The environment refers to the interface that the agent is acting upon it . It may be a maze, a map or any virtual objects like a game play. An agent on the other hand can be a robot, a navigator or a game bot that automatically plays the game using an RL algorithm. A simple architecture of an RL setup is shown in Illustration 1.

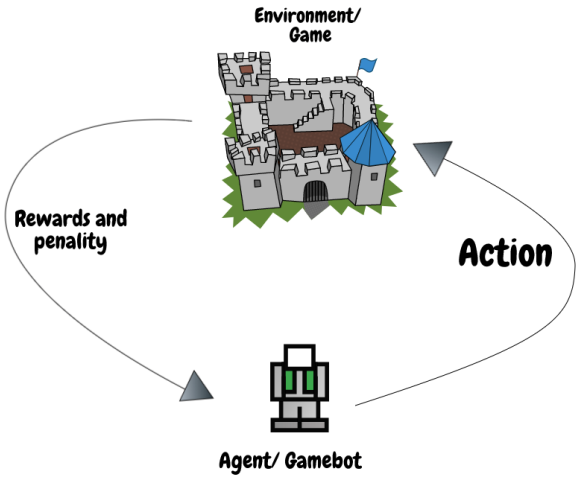

Illustration 1: RL Architecture Diagram

[1] A game's goal is the just another state that is associated with a game. It need not mean a game has to end, but just the state that makes a user content. Example. Some arcarde games has days of game play, and user will be content with just reaching a particular level/score points.

For any good agent that uses RL to reach its goal, should look for maximizing this reward. A typical RL agent works by exploring the environment by taking an action, which leads to it earning Rewards/Penality from the environment. The measure of the reward/penality must be proportional to how good/bad the action that was taken. These measures are fed back to the agent, that uses it for taking a smart next move/action. This step is called as exploitation. Any RL should be able to make a proper trade off between exploration and exploitation.

Exploration vs Exploitation

Exploitation means probing a region of search space that is limited however more promising. In our reinforcement learning realm, it is the optimized actions that we take so that the reward that we collected will be maximum. However, there are poor actions that gives negative or poor rewards initially and there are some chances(probability) that it leads to very high rewards. Thus an RL agent should be able to take the risk, based upon the the probability of considering sub optimal actions once in a while. This is what is called as exploration. The trade off between exploration and exploitation is a whole topic on its own in reinforcement learning.

A Reinforcement Learning agent interacts with the environment in discrete time steps and in doing so it gets values of observations from the environment which includes the rewards(discussed in section 3.1). In the next section we will be discussing the various strategies, that are popular in reinforcement learning that helps the agent to maximize these rewards and to attain the goal.

2.1. Classification of RL Strategies

Model Based vs Model less strategies

Any RL algorithm requires logic of how an agent gets from one state to another. This passage can be said as a probability P that says how likely the agent is to make the move. This probability again depends on the costs/gains that are associated with the move and how good the utility/reward looks in the future. These transitions are associated with each state and action which forms our model. The decision that the agent takes using these parameters is what is called as Markov's decision process[4]. One drawback in such a strategy, is when the state space scales to larger values, then the data(state-action transition) required for building the model also grows, making it computationally hard.

On the other hand Model less strategies, collects information on the fly by exploring randomly in the environment and building data with what it just explored. This reduces the complexity by not including data that are not going to be utilized by the algorithm.

On Policy vs off Policy

Policies are mapping between a state/observation to the action. An RL agent uses the reward values to improve its policies over time. In on-policy, an agent learns a value based on the current action in the current policy. Whereas, in off-policy, the agent need not learn the value from the current action. It may be due to an action that is based on the future or from the past.

2.2. Various Strategies

Q learning

This algorithm relies on a table of values where each value is called a Q value. Each Q value is nothing but the quality of a given action for a given state. Thus the tables are an n-dimensional array of states vs actions where each state comprises of n observations.

The Q learning algorithm is a model-free ,off-policy one, which depends on the optimization equation called Bellman equation [5](shown in illustration 2).

$$V(x) = \max_{a \in \Gamma(x)} \{F(x,a) + \beta V(T(x,a))\}.$$

Illustration 2: Bellman Equation

The Bellman equation[11] depends on both immediate results for taking an action(a) on a state(x) which is F(x,a) and the future value V of the next state(T(x,a)) that was due to taking the action a on x. The future values are subject to discount factor that tells how much the value replies on next value.

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Illustration 3: Q learning algorithm

As seen in illustration 3, the q learning algorithm deals with finding the Q value based on old Q value, the learning rate, reward for making the action on that state the discount factor of the future value. We can see from both illustration 2 and 3 that, how Q-learning equation is related to the bellman equation.

From the figure(illustration 3), The learning rate is a value between 0 and 1, which is tweaked for balancing the time for convergence vs the precision. If the learning rate is high means the algorithm trains faster but with less precision and for very less values, requires more time/iterations to train, though the outputs are precise. It is intuitive to consider higher values of the learning rate(lR) initially to speed us the training and reduce the lR in the later stages for more precision.

The discount factor is used for scaling how much dominant the future value is in the learning. Discount factor of 0 means the algorithm runs on a greedy approach by using the current values alone. And a value of 1 results in full optimization of the future value. For a scenario where a proper goal is not specified, the algorithm with discount factor as 1,will try to assess infinite values, that might cause new errors or the algorithm to malfunction. Hence it is rational to set the discount factor lesser than 1.

The future value that we are considering should be an optimal value. Thus we use the argmax to use the maximum value that is possible in the next state(or min if it is based on penalities), and on continuing this across various iteration, current state will be dependent on optimal next state that is dependent on the optimal state after it. As we appy multiple iterations, we can see that, each Q-value becomes dependent on all the iterations that follow, which makes the Q learning algorithm to get the maximum future reward after adequate number of iterations (The iterations helps the Q value to converge). Q-learning is a variant of value iteration Markov's Decision process strategy that uses transitional probability for convergence.

Policy Iteration MDP

The markov decision process(MDP) is a statistical structure that is used for making a decision where the outcomes of situations are partially random and partially under the discretion of the user.

Policy Iteration[12] consists of multiple iterations that further breaks down into two processes. Policy Evaluation and Policy Improvement.

Policy Evaluation :

$$V(s) := \sum_{s'} P_{\pi(s)}(s,s') \left(R_{\pi(s)}(s,s') + \gamma V(s') \right)$$

This finds the value corresponding to the policy that is based on a greedy policy on the last policy improvement value. This step is performed once for a particular state. Here $P_{\Pi(s)}$ is the probability of the action in the optimal policy in the previous state in the transition from state s to s'. V is the value associated with next state s' and R is the reward associated with the taken action

Policy Improvement :

$$\pi(s) := \arg\max_a \left\{ \sum_{s'} P_a(s,s') \left(R_a(s,s') + \gamma V(s') \right) \right\}$$

This will consider all the actions possible at a given state and picks the own which gives the maximum reward associated when stepping into the next possible states.

The above two processes(Evaluation and Improvement) is executed one after the other in tandem till the value converges.

Value Iteration MDP

The Value iteration counter part for the above is almost similar, except now it is based on only one step. The Π(s) is not used, but is used as a substitution in the V seen above. The value iteration formula is as

$$V_{i+1}(s) := \max_a \left\{ \sum_{s'} P_a(s,s') \left(R_a(s,s') + \gamma V_i(s') \right) \right\},$$

where i+1 is the next iteration whose value depends on the computer value on the previous iteration's optimal(max) value, that was obtained by taking all the actions in reaching next state from state s.

As seen both the policy iteration and the value iterations depends upon the markovian process, in that it is dependent on the transitional probability from state to state. One of the main concerns regarding this is that, it is vulnerable to scalability. Whereas, Q-learning rectifies this problem by introducing randomness in selecting a state/policy and selecting the optimal action that has the best future reward outcome.

SARSA

SARSA stands for State-Action-Reward-State-Action[13]. Just like the acronym suggests, SARSA stands for a five-tuple valued algorithm $(s_i,a_i,r_i,s_{i+1},r_{i+1})$ where s_i is the current state and a_i is the action taken in the current state and r_i is the reward incurred. The action lands the agent in a new state s_{i+1} and the agent chooses another action a_{i+1} in it. SARSA learns the value based on the action that it takes on the current policy (unlike Q-learning which uses an optimal policy), thus making it a **on policy strategy.** In fact the only difference between SARSA and Q-learning is the presence of the optimal policy selected for finding the Q-value for a given state-action combination.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$$

Q value is nothing but the immediate reward which is combined with the future reward(next step) with a factor of the learning rate. Like we saw in Q-learning the value function of Q converges when taking multiple iterations.

Deep Q Networks

The Deep Q Networks is also similar to Q-learning, in that we use deep learning techniques to update the Q-table. The main reason to use this approach rather than Q-learning is that, in the conventional Q-learning we require the number of states to be well-known in prior. In other words, Q-learning doesn't

have an answer for unknown states. This is a problem in partially observable environments.

Another Problem in Q-learning is not in all environments, rewards are instantly available after taking actions. There can be delayed rewards which should be accounted if we need to find the optimal solution.

The problem in Q-learning is rectified by the use of neural networks in Deep Q Networks[6]. Here neural networks are used forestimating the Q value. The input for the neural network is the current state, where as the output is the Q value for the action taken at this state. More formally, it uses a deep convolutional neural network to approximate the optimal action-value function[7]

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \,|s_t = s,\, a_t = a,\, \pi\right],$$

which is the maximum sum of rewards r_t discounted by γ at each time-step t, achievable by a behavior policy Π = P(a/s), after making an observation (s) and taking an action (a)

3. IMPLEMENTATION

For the purpose of implementation we will be considering one of the well-known strategies that we discussed earlier- Q learning. Though it has its drawbacks, it is less complex and can serve as a better platform for learning reinforcement learning in practice. An additional interest is that, it doesn't require much computational power, thus making the implementation easier. In the following sections I have discussed the frameworks for implementing Q learning and the actual game that will be used in the implementation.

3.1. Implementation Framework

Like we saw in any RL to work, we need an environment which gives feedback to the agent(in our case the game bot) in terms of information(Observation) about the environment. It should also give reinforcements like reward/penalities for the agent for making the action to be in that particular environment state.

Since the environment which we are referring to here is the game play, the observations that are desired here are the pixel data of the screen. The pixels represent the state of the game, which is shown. However, the pixels itself cannot represent any information. Example : In the all famous PAC-man[8] game, Just knowing the pixels doesn't help the user infer anything. The Pixel representing the Pac-Man, should be recognized as the protagonist and the pixels of the other colored villains are distinct. Also the position of each objects should be made available., for representing the state along with other parameters.

Apart from the observations available, the user should also be vary of the actions that are available for transition from one state to another and also, what a terminal state look like.

The user should also have information regarding the rewards that we talked about. In game world it is usually the score points(which is again pixel information) that displays on the screen or sometimes is based on temporal metrics like, the time for one complete successful game play. All these information are available under python's module called as Gym which we will be seeing in the next sub section.

Python's Gym Module

Gym is an Openai tool kit that is used for writing and comparing reinforcement algorithms. It is open source and has a collection of hundreds of environments(gaming environments). It also contains the interface to interact with those games, provides the parameters rendered by the game at each stage and also a simple set of actuators for performing actions in the environment.

Using Gym is just an import statement in the Python language. Gym solely supports python and since no other language supports machine learning as good as python, it came as an obvious choice for my implementation. To render a particular game inside Gym, first step is to look at the available games, that Gym supports in the Gym documentation. Then it's just, typing the name of the game in one-line command.

```
env = gym.make('CartPole-v0')
```

command 1

The above command imports the game called cartpole(which is another popular RL driven game) and assigns its environment to the env variable.

The main use of Gym is that, since we normally require image processing tools and an automation interface to make any agent to play a game, Gym helps us in hiding the above complexities and letting us just focus on the algorithm part.

For any game(environment) that is made with the gym command(command1), it has two main spaces: action space and observation space: Action space is just an array of the set of actions that are available for a particular environment(game). For example, If a game supports just the left and right action, the action space contains just two values. Actions are represented as numbers starting from 0. Thus for our two actioned space, we will be having 0 and 1 in the action space; Observation space on the other hand is a vector

of range values. A single state in any game can have multiple observation, and each observation should be within its designated range. For example, If the game depends on the position of the player in the game space, and lets say the positional observation varies from -10 to 10, then each observation that is made during the entire game should be within this range.

Apart from the action space and observation space that is fixed for a game environment, At any instant, after giving the input action, the environment gives back four values

- Observation – This is a set of values that represents the current environment(state). The set itself changes from game to game. For one particular environment, for example it may be just two values – Like position and velocity and for another game it may be something else. These values belong within the range of the observation space.
- The next value returned is the reward. This again keeps changing for different games , where one game may be dependent on positive reinforcements(thereby having a positive reward) and another game on negative reinforcements(thereby having a negative reward). The goal of our RL would be to maximize the total reward achievable.
- The third parameter is the done parameter, which is a boolean representing whether a game episode is done or not done. An episode in a game is when a game reaches a done state. Each episode can contain a number of states that are reached by taking a sequence of actions until termination of the game occurs. It can be a successful termination(winning the game) or an abrupt termination(when the player loses in the middle).
- The final parameter is the info parameter, which gives a lot of other details about the game run for debugging purposes.

The gym also comes with an installed docker container for simulating the whole game play, so the user can visualize how the game bot runs the game after learning the game.

*3.2. **Game setup***

The whole game implementation is dependent on the Q-learning algorithm that we discussed earlier. The game that I have implemented for this paper is **Mountain Car[9] (shown in illustration 4).** It is a classic atari game where the car has to be backed up to the right amount before moving forward, and it cannot move back a lot or forward because of the gravity that is acting upon it.

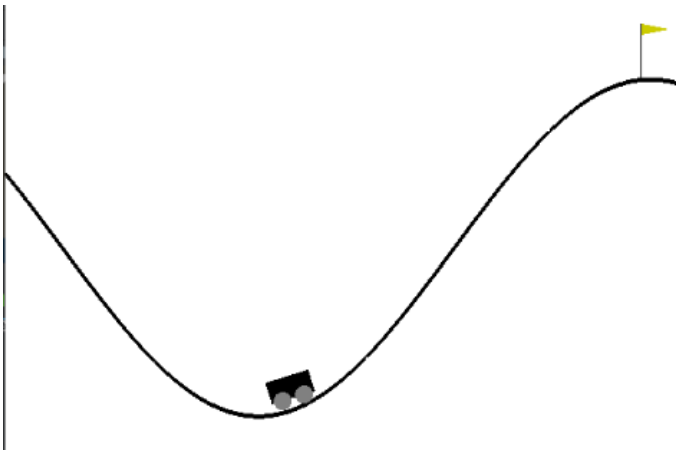

illustration 4

This is a one dimensional game, in that it comes with just three actions in the action space : push left(0), no push (1) or push right(2). The observations in the game is also just a space of two values : position of the car and the velocity with which it moves. Both these observations has a range with which it is associated.

The goal of the game is to move the car back and forth so that the car moves to the top of the mountain with the flag. Since the flag is on the right mountain(as seen in illustration 4), One might suggest the car has to move right alone. But that doesn't work,as at one point on the slope the car loses all its velocity owing to the gravity. Hence the car has to move in the opposite direction to get additional velocity with the help of gravity and use this to propel forward.

The reward associated(or rather the penalty) for each move it makes -1 point. Thus to have an optimal solution, we have to take the minimum number of steps(collect less penalty) to reach the goal,which is the top of the mountain with the flag(in the observation space it is represented with the position value : 0.5)

The Q learning algorithm needs a Q-martix between states and values. Since, the observation has 2 values to represent a state, we have a 3-dimensional array instead of a matrix. The array dimensions are : position and velocity for the state, and action parameter.

Since the position and velocity are continuous value within a fixed range, we bucket the range into discrete values. More the number of discrete state, the faster is the rate of convergence, however we require more space for representing these discrete states. Thus its a trade off between the memory and time. For this experiment, I have chosen the number of states = 40, meaning we will be having 40 discrete positions and 40 discrete velocities and 3 actions allocated to each state (40* 40 * 3 = 4800 state space).

*3.3. **Game play***

The bot starts with at an initial state that was given to it by the gym environment and then it takes a random action. The reward value that was taken in that state is returned by the environment. The bot then continue to use the next optimal action (calculated according to Q learning) for the next step and continue updating the Q table with the rewards collected. This process is repeated till we reach the termination condition. Once the termination condition is reached, we have completed an episode.

For finding the optimal way of solving the problem(collecting very less penalties) means to run a lot of iterations of episodes, each using the previously learnt Q values , till the values converges. I used around 10000 iterations for the Q values to converge. The results of various Iterations are shown in the next section.

Once the Q values converges, the final simulation is as simple as looking for the minimum penalty action in any given state, starting from the initial state, and applying that action in that state. This process of applying the actions, continues till we reach the goal state. According to Q-learning, This will ensure that the game is completed with the minimum penalty collected.

4. RESULTS

As we talked before Q learning is done by using the Q tables values to be updated in an iteration and allowing adequate iterations for the q values to converge.

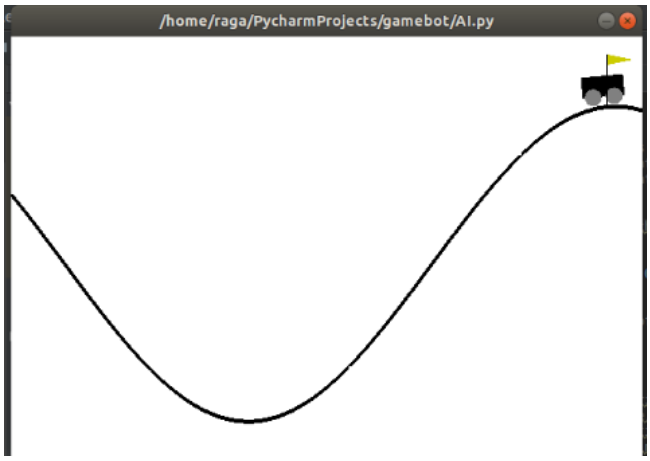

illustration 5

For our Mountain Car example, Each episode (termination of the game play) is completed when the car makes 200 moves(actions) or the car reaches the top of the flagged mountain(illustration 6). If 200 moves are made, it means the bot collects -200 for a particular episode. If a bot is able to reach the top before making 200 moves, then the penality collected for an episode is reduced.

The goal of our Q learning algorithm is to minimize the number of actions that are required for the car to reach the top of the flagged mountain. For proper analysis of the number of iterations(episodes to run), that are needed for the Q values to converge, we started with 100 iterations and increment in proper steps all the way to 10000. In each instance, we find the average score(penalty) obtained across all the episodes(illustration 6).

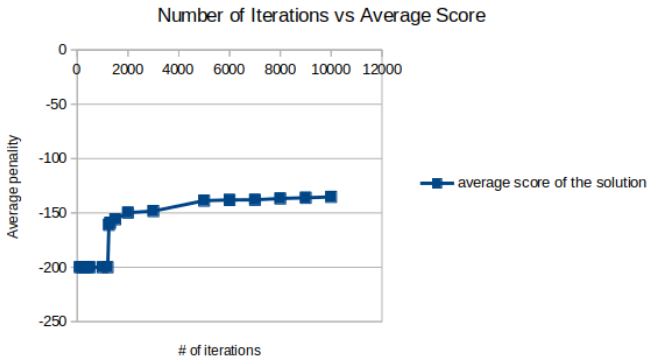

illustration 6

As seen in the above graph, we can see that all the iterations that are lesser than 1250, has an average reward of -200, meaning the episodes has ended, because it has used all 200 moves and stopped as it couldn't find the goal . After using 1250 iterations, the data in the Q tables starts to converge and we can see that the game was able to complete and got to do it in less number of steps(there by lesser penality). Around 10000 iteration we see that the penality we receive is around 130 in average, which means the a lot of episodes was able to complete in an average of 130 steps, rather than running out of 200 steps.

5. CONCLUSION

Q learning is one of the famous research areas in reinforcement learning that doesn't depend on any models and is an off-policy agent. Also we don't need any transitional probability associated from state to state that is required in other reinforcement learning. The implementation that we saw serves as a mirror of how RL works in general.

Any RL algorithm doesn't need any predefined data for modeling the learning algorithm. It figures it by exploring, given the laws of the environment and collecting the information. Once the exploration is done, it tries to find the optimal way of doing it by exploiting the data. This classical case of exploration vs exploitation is what makes an RL algorithm to work properly.

REFERENCES

[1] Kang, A., Jeong, S., Mohaisen, A. and Kim, H. (2016). Multimodal game bot detection using user behavioral characteristics. *SpringerPlus*, 5(1).

[2] Auslander, B., Lee-Urban, S., Hogg, C., Mu˜noz-Avila, H.: Recognizing the Enemy: Combining Reinforcement Learning with Strategy Selection using Case-Based Reasoning.

[3] Hado van Hasselt and Arthur Guez and David Silver: Deep Reinforcement Learning with Double Q-learning

[4] Alagoz, O., Hsu, H., Schaefer, A. and Roberts, M. (2009). Markov Decision Processes: A Tool for Sequential Decision Making under Uncertainty. Medical Decision Making, 30(4), pp.474-483.

[5] *Bellman, R.E. 1957. Dynamic Programming. Princeton University Press, Princeton, NJ. Republished 2003: Dover*

[6] *Volodymyr M., Koray K., David S., Alex G., Ioannis A., Daan W., Martin R. : Playing Atari with Deep Reinforcement Learning*

[7] *Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), pp.529-533.*

[8] Schiesel, Seth (2007-06-06). "Run, Gobble, Gobble, Run: Vying for Pac-Man Acclaim". The New York Times. Retrieved 2011-02-08

[9] *[Moore, 1990] A. Moore, Efficient Memory-Based Learning for Robot Control, PhD thesis, University of Cambridge, November 1990.*

[10] *Tejas D., Karthik R.,Ardavan S ,Joshua B.:Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation*

[11] Anon, (2018). *DQN Getting Started to Abandon 4 Dynamic Planning and Q-Learning*. [online] Available at: https://zhuanlan.zhihu.com/p/21378532?refer=intelligentunit [Accessed 25 Apr. 2018].

[12] Blog.csdn.net. (2016). 增强学习 *Reinforcement Learning* 经典算法梳理 *1*：*policy and value iteration - CSDN* 博客. [online] Available at: https://blog.csdn.net/songrotek/article/details/51378582 [Accessed 25 Apr. 2018].

[13] Online Q-Learning using Connectionist Systems" by Rummery & Niranjan (1994)