International Conference - Innovation-2021-Innovation-2021
International Journal of Scientific Research in Computer Science, Engineering and Information Technology ISSN : 2456-3307 (www.ijsrcseit.com)
Volume 8, Issue 3, May-June-2021

Training an Agent using Deep Reinforcement Learning: Snake Game

Kartik Kaushik[1], Reetej Chindarkar[1], Rutuja Vetal[1], Ronak Thusoo[1], Prof. Pallavi Shimpi[2]
[1]Student, Department of Computer Engineering, Dr. D. Y. Patil School of Engineering, Pune, Maharashtra, India
[2]Assistant professor, Department of Computer Engineering, Dr. D. Y. Patil School of Engineering, Lohegaon, Pune, Maharashtra, India

ABSTRACT

Deep Reinforcement Learning has become a commonly adopted method to enable agents to hunt out complex control policies in various video games. Deep-Mind used this technique to play Atari games. However, similar approaches should get to be improved when applied to tougher scenarios, where reward signals are sparse and delayed. This paper illustrates a refined Deep Reinforcement Learning model to enable an autonomous agent to play the classical Snake Game, whose constraints get stricter as the game progresses further. Specifically, to train this model we have used Deep Neural Network (DNN) with a variant of Q-learning where agent will learn from its past experiences. Moreover, we have proposed a designed reward mechanism to properly train the network, adopt a training gap strategy to temporarily bypass training after the situation of the target changes, and also introduces dual experience replay method through which different experiences for better training can be categorized. The final results show that our agent in an environment outperforms the baseline model and surpasses the human-level performance in terms of playing the Snake Game.

Keywords: Deep reinforcement learning, Q-Learning, Deep Neural Network, Deep Learning, Experience replay.

I. INTRODUCTION

Neural Networks when combined along with the reinforcement algorithms can beat human experts playing various Atari video games. Deep-mind's AlphaGo, an algorithm that had beaten the world champions of the Go board game. At DeepMind they pioneered the mixture of these approaches i.e. deep reinforcement learning - to form the first artificial agents to understand human-level performance across many challenging domains.

Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward during a particular situation. It is employed by various software and machines to seek out the simplest possible behaviour or path it should absorb a selected situation. Reinforcement learning differs from the supervised learning during a way that in supervised learning the training data has the solution key with it therefore the model is trained with the correct answer itself whereas in reinforcement learning, there's no answer but the

reinforcement agent decides what to try to perform the given task. It is bound to learn from its experience. Reinforcement learning also differs from the unsupervised learning. Where unsupervised learning deals with associative rule mining and clustering and on the other hand Reinforcement learning deals with exploration, decision process of Markov, Value learning, deep learning and Policy learning. Unsupervised learning deals with the data which is unlabelled where output is based on some perception or collection of perception. As name suggests unsupervised learning is not supervised and reinforcement learning is less supervised which is totally dependent on the agent identifying the output. To summarize in supervised learning, we generate formula based on input and output we provide. In unsupervised learning we find the relation or association between input and output values. In Reinforcement learning agent learns using delayed feedback by communicating with the environment. In this paper, our agent learns how to play the snake Game by interacting with the environment. Agent choose some action get feedback from environment. The feedback is in the form of states or rewards. This cycle is continued till our agent end up in the terminal state. Then learning of new episode starts.

Episode is the length of simulation. At the end of simulation system end up in terminal state. We have used Deep Q-Network (DQN). DQN is known to be first step of Reinforcement learning. DQN is reinforcement learning algorithm that combines deep learning neural networks with Q-learning to let Reinforcement learning work for high dimensional, complex environments like video games, or robotics. So, we rely specifically on deep Q-learning network (DQN) that chooses the best action based on both the observations i.e. from the environment & prior learned knowledge to train an agent, In order to successfully learn to play this Snake Game is quite challenging because the restrictions of this AI Snake Game gets stricter & stricter as the snake grows in length & the game gets going. Also, to add, an apple once it is eaten by the snake, using random function a new one is immediately spawned at a random location.

This is changing target issue. We have studied various techniques & used the best ones to handle this issue. You can see the results between the DQN model and human level performance. this performance can be viewed in terms of time or score. After that total reward can be calculated by network. The way humans learn by using their memory from past experiences, similarly DQN uses this technique too. Experience reply and reply memory are part of this technique. Experience reply allows our agent to store or memorize along with reusing the past experiences just as humans tend to replay crucial experiences and generalize them to the situation at hand. Replay Memory is like a stack which stores the agent's experiences and it is mainly used to train the DQN. We will be more focusing on the learning and training the agent rather than the game.

II. METHODS AND MATERIAL

A. Deep Reinforcement Learning

Deep reinforcement learning combines artificial neural networks with a reinforcement learning architecture that permits software-defined agents to learn the best actions possible in a virtual environment to realize their goals. It unites function approximation & target optimization and it maps state-action pairs to get expected rewards.

The network exists of layers with nodes, the primary layer is that the input layer. Then the hidden layers will rework with all the information along with weights and activation functions. The last layer is the output layer, where the target is expected. Adjusting the weights will help the network to learn patterns and improve its predictions.

B. Q-Learning

Q-learning is a model-free reinforcement learning algorithm [1]. Q-learning is a values-based learning algorithm. Value based algorithms updates the value function based on an equation (particularly Bellman equation) [1]. Whereas the other type, greedy policy obtained from the last policy improvement is estimated by policy-based value function [1].

Here are some definitions which are used in Q-Learning:

- Q*(s,a) is the expected value (cumulative discounted reward) of doing a in state s and then following the optimal policy[1].
- Temporal Differences (TD) is used by Q-learning uses to estimate the value of Q*(s,a). Temporal difference is an agent learning from an environment through episodes with no prior knowledge of the environment [1].
- The agent maintains a table of Q [S, A], where S is the set of states and A is the set of actions [1].
- Q [s, a] represents its current estimate of Q*(s,a) [1].

Q-Table is a data structure that guides us to the best action at each state. Q-Learning algorithm is used to learn each value of the Q-table.

Bellman Equation is used for Q-function and it takes two inputs i.e. State (s) & Action (a).

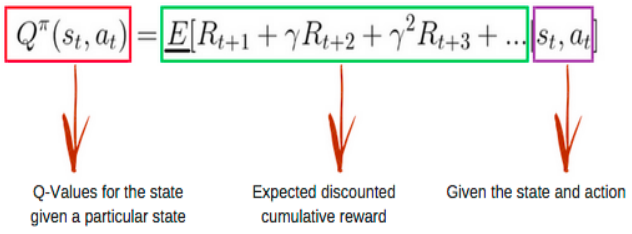


Fig 1. General Bellman Equation

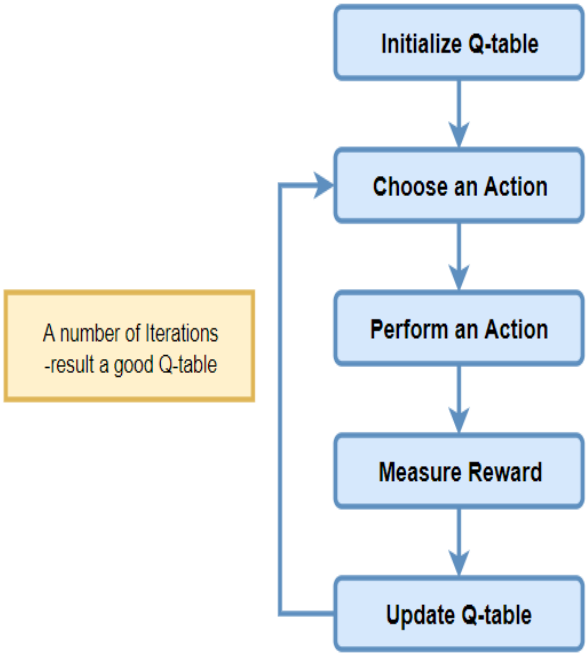


Fig 2. Q-learning Algorithm Process

The steps that Q-Learning Algorithm follows are:

Step 1: Initialising the Q-Table

First the Q-table has to be built. The table is divided into 'n' columns & 'm' rows, where 'n' is the number of actions and 'm' is the number of states.

Step 2: Choosing an action

In this stage, the agent will choose an action to perform

Step 3: Performing an action

During this stage, steps 2 & 3 will be performed for an undefined amount of time. Initially, an Action (a) in State (s) is chosen according to the Q-Table. Every Q-value is zero when the episode initially starts. The Q-values are then updated according to the Bellman Equation.

Epsilon greedy strategy concept is used here. Initially, the epsilon rates are higher & the agent explores the environment and randomly chooses actions to perform. This happens logically, since the agent does not know anything about the environment. As the

agent starts to explore the environment, the epsilon rate will decrease and the agent will then start to exploit the environment. The agent becomes more confident in estimating the Q-values, as the level of exploration done by the agent increases. [5]

Step 4: Measure Reward

In this stage, we measure the reward by observing the outcome based on the action taken.

Step 5: Evaluate

In this stage, the function Q(s,a) is updated. This process will be repeated till the learning process is completed. In this way, the Q-table keeps getting updated & the value function Q will get maximised. Here, Q(s,a) returns the expected future reward of the action performed in that state.

Initially, to update the Q-table we will explore the environment using the agent. After the Q-table has been finished updating, the agent will start exploiting the environment & will start taking better actions.

C. Deep Q-Network (DQN)

A DQN, or Deep Q-Network, approximates a state-value function in a Q-Learning framework with a neural network [2].

Here are some definitions that are used in DQN:

Agent: An **agent** is something that takes actions.

Action (a): 'a', it is the set of all the possible moves that an agent can make. An **action** is something that the agent chooses from a discrete list of possible actions. In our case, the agent can choose from the moment set of left, right, up and down.

Discount Factor: The **discount factor** is multiplied by future rewards as discovered by the agent in order to dampen these rewards' effect on the agent's choice of action [3]. Discount factor is designed to make immediate rewards more significant than future rewards. It is expressed with the lower-case Greek character, gamma: γ.

Environment: **Environment** can be described as the world that the agent belongs in and the world that responds according to what the agent does. The environment will take the agent's current state and action as input, while giving the agent's reward and its next state as the output.

State (s): A **state** can be described as the current and the immediate situation of the agent i.e. a specific place or moment, any configuration that puts our agent in relation to obstacles and prizes.

Reward (r): A **reward** is the feedback by which we measure the success or failure of an agent's actions in a given state [3]. It is the measurement of whether the agent's action in that given state were successful or not. Rewards can be immediate or delayed.

Policy (π): The **policy** is defined as the strategy that the agent employs to determine the next action based on the current state. It maps states to actions that promise the highest reward.

Value (v): **Value** is the expected long-term return with discount, when compared to the short-term reward 'r'. 'vπ(s)' is defined as the expected return of the current state & the policy 'π'.

Q-value (Q): **Q-value** is almost similar to Value but it takes an extra parameter i.e. action 'a'. Qπ (s, a) refers to the expected return which includes the action 'a', policy 'π' & the current state 's'. Q-value maps state-action pairs to rewards.

DQN overcomes unstable learning by mainly using four techniques:

Experience Replay:

It is hard to produce various experiences, once DNN is overfitted. In order to overcome this problem, Experience Replay stores experiences including state transitions, rewards and actions, which are necessary data to perform Q learning. It also makes mini-batches to update the neural network [4]. This technique has the following advantages:

- It will reduce the correlation between experiences when updating the DNN.
- It will also increase the learning speed with the help of mini-batches.
- It will reuse past transitions to avoid catastrophic forgetting.

Target Network:

Target function is changed frequently with DNN while calculating TD error. Training data could be difficult when using unstable target function. Target Network technique is used to fix the parameters of target function and replace them with the latest network every thousand steps.

$$Q(s_t,a) \leftarrow Q(s_t,a) + \alpha \left[r_{t+1} + \gamma \max_p Q(s_{t+1},p) - Q(s_t,a) \right]$$

Fig 3*. Target Q-function in the red box shown is fixed*

Clipping Rewards:

Clipping Rewards technique is used to clips scores, according to which all positive rewards are set +1 and all negative rewards are set -1 [6, 7].

Skipping Frames:

Skipping Frames technique is defined such that the DQN calculates the Q-value every four frames and uses the past four frames as input which in return reduces the computational cost and helps in gathering more experience [6, 8].

Algorithm: Deep Q-learning with Dual Experience Replay

Requirements:

- MP_1 and $MP2$ = replay memory functions.
- N = Memory Pool for storing the experience replay.
- a_t = Action done at time t.
- s_t = State at time t.
- r_t = Reward at time t.
- e_t = Experience at time t.
- η (eta) = sampling proportion.

Initialize replay memory MP_1 & $MP2$ to $N/2$
Initialize Q with random weights

for all training steps **do**
Initialize state $s1$ for the new episode
Pre-process $\varphi 1 = \varphi(s1)$
repeat
Using Exploration probability ε select a random action at
otherwise select $at = argmax_a A Q(\varphi(st), a)$
Decay exploration probability ε
Execute at in game then observe rt and $st+1$
Pre-process $\varphi t+1 = \varphi(st+1)$
if $|rt| \geq 0.5$ **then**
Store $et = (\varphi t, at, rt, \varphi t+1)$ in MP_1
else
Store $et = (\varphi t, at, rt, \varphi t+1)$ in MP_2
Sample mini-batch of ek from $MP1$ & $MP2$
In segments of η and $(1 - \eta)$ respectively
Decrease sampling proportion η
if episode terminated at $\varphi k+1$
Target value $Q^*_k = rk$
else
$Q^*_k = rk + y max_{a \in A} Q(\varphi k+1, a)$
Define loss function $loss = (Q^*_k - Q(\varphi k, ak))^2$
Update neural network parameters by performing optimization algorithm Adam on $loss$
until episode terminates
end for

Actions	
Snake moves up	0
Snake moves right	1
Snake moves down	2
Snake moves left	3

Rewards	
Snake eats an apple	10
Snake comes closer to the apple	1
Snake goes away from the apple	-1
Snake dies (hits his body or the wall)	-100

State	
Apple is above the snake	0 or 1
Apple is on the right of the snake	0 or 1
Apple is below the snake	0 or 1
Apple is on the left of the snake	0 or 1
Obstacle directly above the snake	0 or 1
Obstacle directly on the right	0 or 1
Obstacle directly below the snake	0 or 1
Obstacle directly on the left	0 or 1
Snake direction == up	0 or 1
Snake direction == right	0 or 1
Snake direction == down	0 or 1
Snake direction == left	0 or 1

Fig 4. Action, Reward State as we have defined.

Different types of state space that we have used:

Maybe it's possible to change the state space and achieve similar or better performance than when the agent learns to play snake using experience reply. The following four state spaces were tried:

- State space 'no direction': The agent was not given the direction the snake was going in.
- State space 'coordinates': We replaced the location of the apple (up, right, down and/or left) with coordinates like apple (x, y) and the snake (x, y). The coordinates can be scaled between 1 and 0.
- State space 'direction 0 or 1': This is the original state space.
- State space 'only walls': We did not give the agent the direction of the body, only tell it if there's a wall.

```
# epsilon sets the level of exploration and decreases over time
param['epsilon'] = 1
param['epsilon_min'] = .01
param['epsilon_decay'] = 1/100

# gamma: value immediate (gamma=0) or future (gamma=1) rewards
param['gamma'] = .95

# the batch size is needed for replaying previous experiences
param['batch size'] = 500

# neural network parameters
param['learning_rate'] = 0.00013629
param['layer_sizes'] = [128, 128, 128]
```

Fig 5. Parameters Defined for DQN

Epsilon (ε) defines the probability of exploration & it will decrease as the agent explores the environment more & more.

Gamma is also known as the discount factor. It is designed to make future rewards less than the immediate rewards.

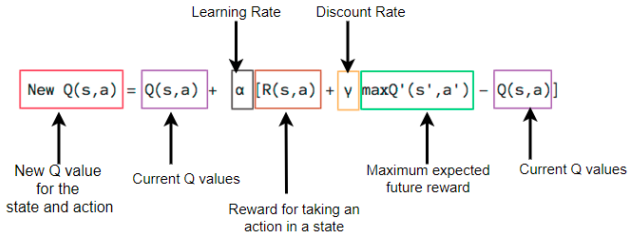
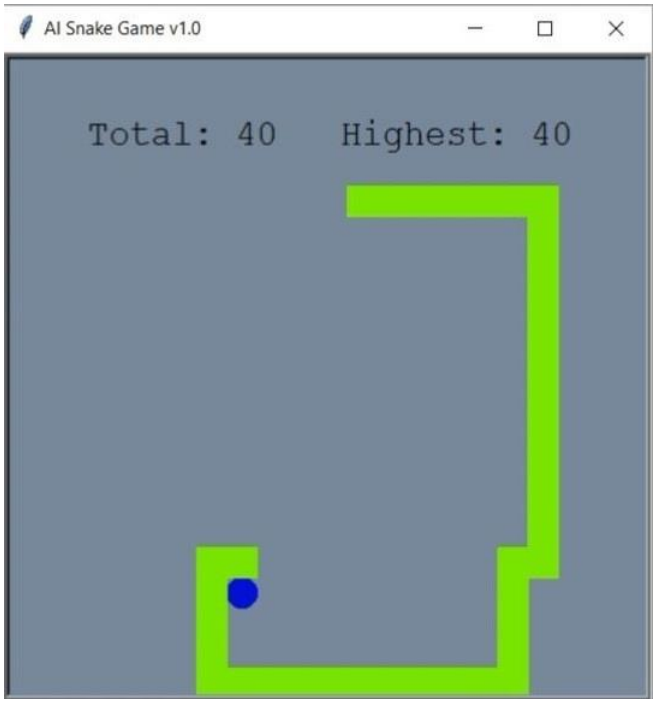


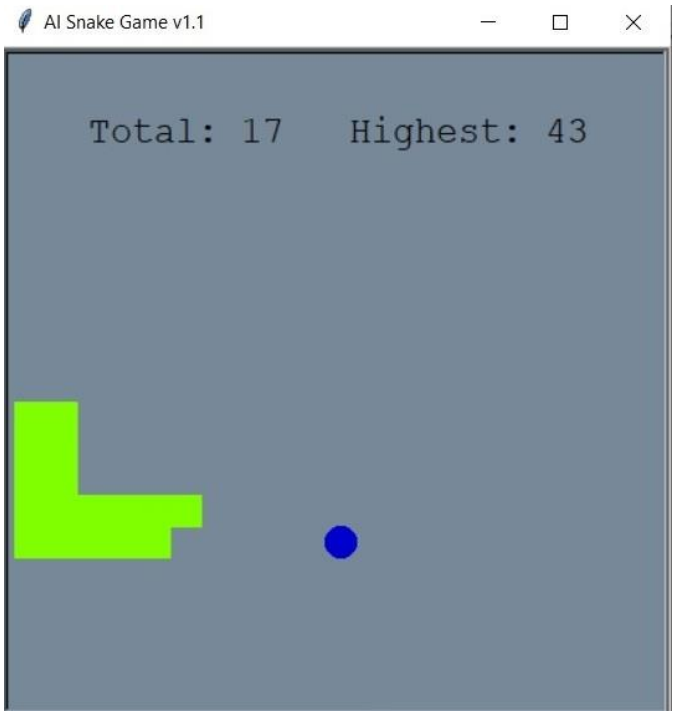
Fig 6. Bellman Equation

III. EXPERTIMENTAL RESULTS

Firstly, we decided to use different parameters for our AI Snake Game by creating two different versions i.e. v1.0 & v1.1. In these versions, the basic difference was only of the parameter 'Epsilon Decay' & 'Learning Rate'. We used the values '.995' & '1/100' for Epsilon Decay respectively. According to our observation, the epsilon decay with value '1/100' had better results when compared to epsilon decay with value '.995'. We also used the values '0.00025' & '0.00013629' for the learning rate in our Neural Network.



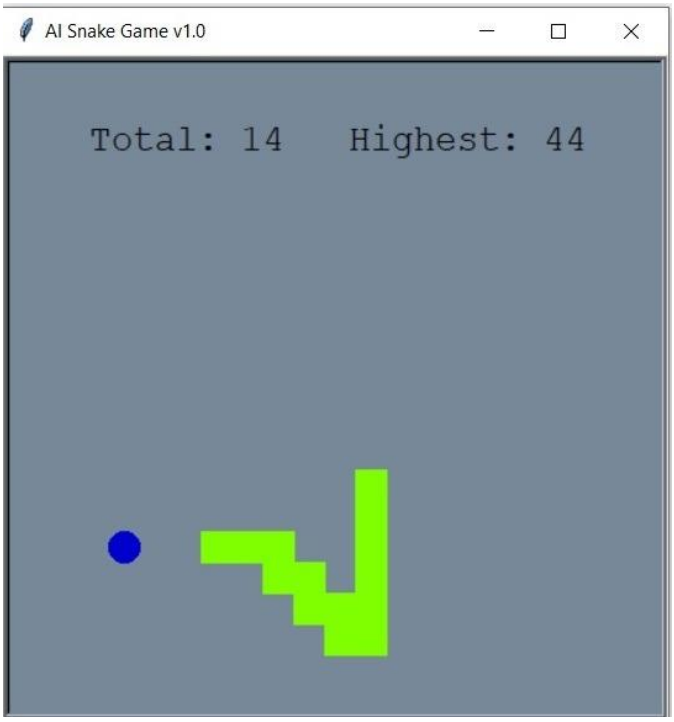
(a) Version 1.0



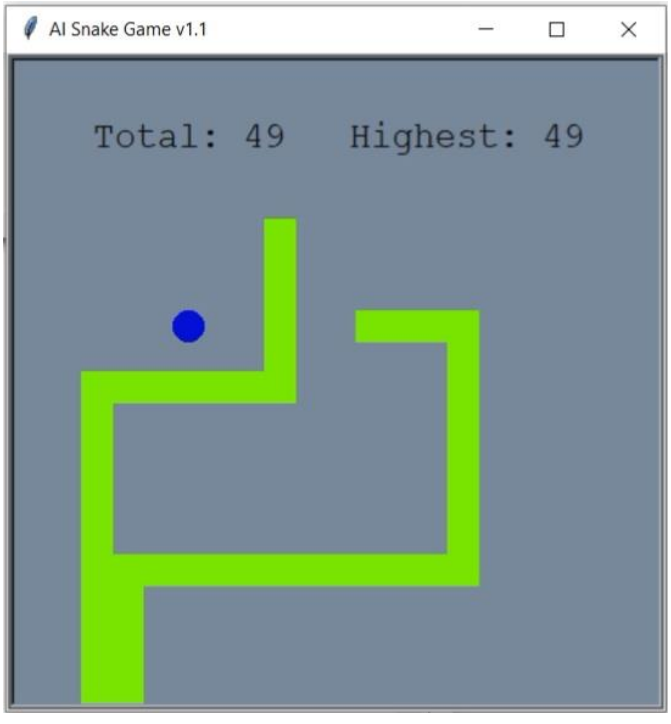
(b) Version 1.1

Fig 7. High Score of 50 Episodes

According to our observation, version 1.1 of the AI Snake Game which has the values of 1/100 & 0.00013629 for Epsilon Decay & Learning Rate respectively performed better when compared to version 1.0 of AI Snake Game. The highest score observed in version 1.0 was 40 for 50 Episodes (E), similarly for 50 Episodes v1.1 observed 43 as the highest score.



(a) Version 1.0



(b) Version 1.1

Fig 8. High Score of 100 Episodes

After observing that 50 Episodes were not enough to train our agent, so the number of episodes were increased to 100. As can be seen in the figures above, version 1.0 only had an increase of four points in the highest score while version 1.1 had an increase of six points.

IV. CONCLUSION

In this paper, we have tried to implement the classical snake game using Deep Reinforcement Learning & DQN (Deep Q-Network), while also using python libraries like turtle, seaborn, TensorFlow, NumPy, Keras, etc. We have used Adam optimization for our deep learning model, as it helps in faster convergence. Improper training experiences have been eliminated and proper function of agent has been done which helps in better performance with increase in levels. It will provide relatively better results as when compared to the existing techniques.

As we have observed our model still has some issues like enclosing problem where the Snake cannot see the whole environment and the agent will enclose itself and die. This issue was observed especially when the Snake was of larger length.

To solve the issue of enclosing we can use pixels and CNN (Convolutional Neural Network) in State Space. Also, we can assign regulated weights for a better model. We can also use Double Deep Q-Learning Algorithm instead of the normal Deep Q-Learning Algorithm to get a more precise convergence. Bayesian Optimization can also be used to further improve the network.

V. REFERENCES

[1]. Chathurangi Shyalika, “A Beginners Guide to Q-Learning,” Towards Data Science, 15 Nov, 2019.

[2]. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” ArXiv e-prints, 2013.

[3]. Chris Nicholson, “A Beginner's Guide to Deep Reinforcement Learning,” Path Mind.

[4]. L.-J. Lin, “Reinforcement learning for robots using neural networks,” Ph.D. dissertation, Pittsburgh, PA, USA, 1992, UMI Order No. GAX93-22750.

[5]. R. S. Sutton and A. G. Barto, “Reinforcement learning: An introduction,” IEEE Transactions on Neural Networks, vol. 9, no. 5, pp. 1054–1054, 1998.

[6]. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness,M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[7]. D. Wang and A.-H. Tan, “Creating autonomous adaptive agents in a real-time first-person shooter computer game,” IEEE Transactions on Computational Intelligence and AI in Games, vol. 7, no. 2, pp. 123–138, 2015.

[8]. H. Y. Ong, K. Chavez, and A. Hong, “Distributed deep Q-learning,” ArXiv e-prints, 2015.