Playing Snake with Reinforcement Learning

Ananth Shreekumnar
PUID: 0034176670

April 2022

Abstract

In this report, we explore several reinforcement learning algorithms and study their performance in the context of learning to play the snake game. We also experiment with various state spaces that model the game as a Markov Decision Process and as a Partially Observable Markov Decision Process.

1 Introduction

Deep reinforcement learning has successfully been used to train agents to play games in the past Silver et al. [2016], Mnih et al. [2013]. The snake game is a challenging environment to model since the constraints get stricter as the game progresses. In fact, any game that involves collecting an item, location traversal, and traversing a one way path is NP-hard Viglietta [2012], which motivates the goal of learning an agent to model the environment rather than use a deterministic algorithm.

Indeed, if each grid point in the game is modelled as a node and each 4 way adjacent node is connected by edges, then the game can be solved perfectly if there exists a Hamiltonian cycle on this induced graph. For such a graph, a Hamiltonian cycle exists only when either the width or the height of the grid are even and it can be computed in polynomial time Chiba and Nishizeki [1989]. However, the deterministic algorithm fails when this condition is not met.

Section 2 elaborates the rules of the Snake game and section 3 lists the configurations of the game that will be studied. Section 4 presents the algorithms that were used to train an agent and section 5 presents the results. We conclude in section 6.

2 Snake

A typical implementation, and one that we use in our experiments, consists of a snake that moves around on a grid. Figure 1 shows a sample of the screen. The green is the snake's body, white is the snake's head, red is the fruit, and black is empty space.

At any step, the snake has 4 actions—left, right, up, or down. These actions are taken on the snake's head. However, one of these actions, the opposite to the current direction, is not allowed. For example, if the snake is moving towards the right, a left move will result in no action being taken. Furthermore, if no action is specified, the snake will continue to move in the same direction it is currently moving.

The goal is to collect as many of the fruits that spawn on the grid. Once a fruit is collected, it reappears in an unoccupied part of the grid randomly. The snake grows by one unit each time it

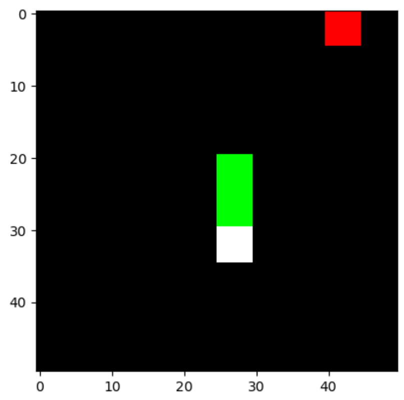


Figure 1: A sample picture of the game screen

collects a fruit. The game terminates when the entire grid is covered by the snake, or when the snake collides—which is defined as the snake moving into a space that is either occupied by itself or is beyond the border specified by the grid.

3 Game Configurations

For the purpose of our experiments, we fix the grid to be a square of width 10 and height 10. This is solely due to the limited computational resources available. Note that each of the algorithms specified in section 4 can be run on a grid of any size.

The action space of this game is straightforward—there are 4 enumerable actions. Since we implement the environment from scratch, we have complete control over what a state entails. Thus, we formulate and experiment with 3 different state spaces:

(i) **Simple**: This state is a binary vector (each value is either 0 or 1) of length 16. The first 8 values represent if each of the 8 directional neighbors of the snake's head is *blocked*. Here, by *blocked*, we mean that the grid point is either out of bounds or is occupied by the snake. The remaining 8 values are a one hot encoding of the 8 directional position of the fruit relative to the snake's head.

(ii) **2D**: This state is a 2 dimensional representation of the grid, i.e., it is a 10×10 matrix with each entry an integer between -1 and 2 whose meaning is given in table 1.

-1	fruit
0	empty
1	snake body
2	snake head

Table 1: What each entry denotes in the 2D state representation.

(iii) **Framestack:** This state is motivated from the fact that a human player does not require a special color for the head of the snake to play the game. Instead, humans reply on the past positions of the snake to derive the direction that the snake is moving. However, simply removing the special encoding for the snake's head from the 2D representation is not enough. Mnih et al. [2013] solved this problem by creating a state that is infact the previous 4 states of the environment stacked as channels, much like the channels of an image. Together, these 4 states encode information about the direction of the snake.

The reward kernel of the game is straightforward. Eating the fruit gets a reward of 100 and collisions give a reward of -100. Otherwise, the reward is the difference in the Euclidean distance between the snake's head h and the fruit f in consecutive time steps given by equation 1.

$$r = \| h_t - f_t \|_2 - \| h_{t-1} - f_{t-1} \|_2 \tag{1}$$

4 Algorithms

We experiment with 2 broad implementations of Q learning, tabular learning Watkins [1989] and and Deep Q Learning Mnih et al. [2013]. We implement and test 2 variants of Deep Q Learning:

(i) **Vanilla DQN:** A simple implementation with 2 essential enhancements elaborated in section 4.2.

(ii) **Double DQN:** This implementation uses a prioritized experience replay buffer Schaul et al. [2015] that samples experiences nonuniformly and performs Double DQN Learning updates as given in section 4.3.

Each of the above algorithms is trained using an ϵ-greedy approach to explore the state space and gather data. We use a schedule that linearly decreases the value of ϵ over time. This is because we would like to explore heavily at the start of training, since the agent has no useful experience. As the agent explores, we graduate let it take its own actions and learn from them. Note that this is just a way of speeding up the training process—the algorithm would still work with a constant ϵ, but would take longer to train. Mathemtically, at time step n, the value of ϵ using this schedule is given in equation 2.

$$\epsilon(n) = \max\left(b - n\delta, m\right) \tag{2}$$

4.1 Tabular Q Learning

In this implementation of Q learning, we maintain a Q value for each state action pair. We then update these Q values as the agent explores the environment as given in algorithm 1. As long as the agent visits each state action pair enough number of times, this is guaranteed to converge to the optimal Q values.

4.2 Vanilla Deep Q Learning

The drawback of tabular Q learning is that it is not practical for continuous state actions spaces and indeed even for discrete state action spaces when their size is too large. The problem arises since tabular Q learning requires each state action pair to be visited enough to converge to a good approximation to the optimal Q values. The Deep Q Learning algorithm Mnih et al. [2013] approximates the table by learning it using a neural network. However, 2 additional enhancements are required in practice:

1. addition of a uniform experience replay buffer—store experiences (state, action, new state, reward, done) in a buffer. At every step, learn from a batch of samples sampled uniformly at random from the buffer.

Algorithm 1 Tabular Q Learning.

```
1: **Input:** learning rate $\alpha$, $\epsilon$ schedule, discount factor $\gamma$
2: Initialize $Q(S, A) = 0$ for each $S \in \mathcal{S}$, $A \in \mathcal{A}$

3: for each episode do
4:     Initialize starting state $S$
5:     Get $\epsilon$ from schedule
6:     while $S$ is not a terminal state do
7:         Choose $A$ using $\epsilon$-greedy policy
8:         Take action $A$, observe reward $R$ and next state $S'$
9:         $Q(S, A) = (1 - \alpha)Q(S, A) + \alpha(R + \gamma \max_{a \in \mathcal{A}} Q(S', a))$
10:        $S = S'$
11:    end while
12: end for
```

2. use 2 identical deep networks while training. One of these is used to get the action, and the other is used to get the Q values of the next state. Weights are copied from the offline model to the online model periodically once in a while.

Both these enhancements help stabilize the learning process:

1. Using a large enough experience replay, each experience may be potentially used to learn from multiple times, which improves data efficiency.

2. Learning on-policy directly from consecutive samples is inefficient due to the strong correlation between samples.

3. Learning using on-policy means that the current parameters determine the next data sample that the parameters are trained on. Then it is possible that the samples are biased and might to lead to unwanted feedback loops which could result in the model parameters being stuck in a poor local minimum or even diverge Tsitsiklis and Van Roy [1997].

This enhanced version of the Deep Q Learning algorithm is given in algorithm 2.

4.3 Double Deep Q Learning with a Prioritized Experience Replay Buffer

The max operator in tabular Q learning and Deep Q Learning makes it more likely to select overestimated values since it uses the same values both to select and to evaluate an action. This could result in overoptimistic value estimates. Double Q Learning Hasselt [2010] is a method to prevent this. Using the double Q update in DQN van Hasselt et al. [2015], the target in line number 13 of algorithm 2 is changed to equation 3.

$$y_j = R_j + \gamma Q(S_{j+1}, \arg\max_{a \in \mathcal{A}} Q(S_{j+1}, a; \theta); \theta') \tag{3}$$

We also use an experience buffer that samples experiences nonuniformly—it prioritizes those samples from which the most learning can be done Schaul et al. [2015]. Particularly, we use the proportional prioritization variant, and this algorithm is given in algorithm 3. Every new experience is first assigned a maximum priority since it has not been used for learning yet. Then, these priorities are adjusted when replayed. The value of α controls the amount of prioritization, with $\alpha = 0$ corresponding to the uniform case.

Algorithm 2 Deep Q Learning.

```
1: Input: learning rate $\alpha$, $\epsilon$ schedule, discount factor $\gamma$, replay memory capacity $N$
2: Initialize $Q(S, A; \theta)$ for each $S \in \mathcal{S}$, $A \in \mathcal{A}$ with random weights.
3: Initialize a copy $Q(S, A; \theta')$
4: Initialize replay buffer $\mathcal{D}$ with capacity $N$

5: for each episode do
6:     Initialize starting state $S$
7:     Get $\epsilon$ from schedule
8:     while $S$ is not a terminal state do
9:         Choose $A$ using $\epsilon$-greedy policy
10:        Take action $A$, observe reward $R$ and next state $S'$
11:        Store experience $(S, A, R, S')$ in $\mathcal{D}$
12:        Sample a minibatch $(S_j, A_j, R_j, S_{j+1})$ from $D$ uniformly
13:        Set $y_j = \begin{cases} R_j & \text{for terminal } S_{j+1} \\ R_j + \gamma \max_{a \in \mathcal{A}} Q(S_{j+1}, a; \theta') & \text{for nonterminal } S_{j+1} \end{cases}$
14:        Perform a gradient descent step on $(y_j - Q(S_j, A_j; \theta))^2$
15:        $S = S'$
16:        From time to time, copy weights into target network $\theta'$
17:    end while
18: end for
```

The use of prioritized sampling is beneficial since the event where the snake eats the fruit is quite rare, thus the reward has a sparse structure. This is partly mitigated by introducing the distance measure to provide small rewards at every time step, but the use of priority sampling alleviates the issue further.

However, prioritized replay might introduce bias since it is free to change the distribution in an uncontrolled fashion. Weighted importance sampling is used to control this bias by correcting the probabilities using equation 4.

$$w_i = \left(\frac{1}{N} \frac{1}{P(i)}\right)^{\beta} \tag{4}$$

We normalize these weights by $\frac{1}{\max_i w_i}$ for the stability of deep learning. In practice, β is linearly annealed from a base value b to 1 given by equation 5. Here, we set δ such that β reaches a value of 1 at the end of training.

$$\beta(n) = \min\left(1, b + n\delta\right) \tag{5}$$

5 Experiments and Results

The ϵ schedule that we used is empirically designed. Figure 2 shows a sample schedule with base $b = 1$, a rate of decay $\delta = 1e - 4$. We stop the decay when it reaches a value of $m = 1e - 3$, which corresponds to 9990 episodes.

The β schedule that we used is a linear growth schedule with the value of β reaching 1 towards the end of training. Figure 3 shows a sample schedule with base $b = 0.6$ and a rate of increase $\delta = 2e - 5$, which means the value of β is 1 after 20000 episodes.

Algorithm 3 Deep Q Learning.

```
1: **Input:** learning rate $\alpha$, $\epsilon$ schedule, discount factor $\gamma$, replay memory capacity $N$, exponents $\alpha$ and $\beta$ schedule for prioritization
2: Initialize $Q(S, A; \theta)$ for each $S \in \mathcal{S}$, $A \in \mathcal{A}$ with random weights.
3: Initialize a copy $Q(S, A; \theta')$
4: Initialize prioritized replay buffer $\mathcal{D}$ with capacity $N$ and $p_1 = 1$

5: for each episode do
6:     Initialize starting state $S$
7:     Get $\epsilon$ and $\beta$ from schedule
8:     while $S$ is not a terminal state do
9:         Choose $A$ using $\epsilon$-greedy policy
10:        Take action $A$, observe reward $R$ and next state $S'$
11:        Store experience $(S, A, R, S')$ in $\mathcal{D}$ with maximum priority $\max_{i \in \mathcal{D}} p_i$
12:        Sample a minibatch $(S_j, A_j, R_j, S_{j+1})$ from $D$ with probability $P(j) = \frac{p_j^\alpha}{\sum_k p_k^\alpha}$
13:        Compute importance sampling weights $w_j = \frac{(NP(j))^{-1}}{\max_i w_i}$
14:        Set $y_j = \begin{cases} R_j & \text{for terminal } S_{j+1} \\ R_j + \gamma Q(S_{j+1}, \arg\max_{a \in \mathcal{A}} Q(S_{j+1}, a; \theta); \theta') & \text{for nonterminal } S_{j+1} \end{cases}$
15:        Compute $\delta_j = y_j - Q(S_j, A_j; \theta)$
16:        Compute priorities $p_j = |\delta_j|$
17:        Perform a gradient descent step on $(w_j \delta_j)^2$
18:        $S = S'$
19:        From time to time, copy weights into target network $\theta'$
20:    end while
21: end for
```

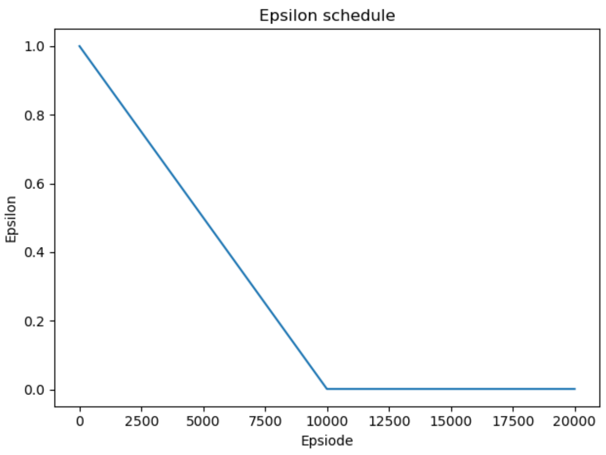


Figure 2: ϵ schedule for training

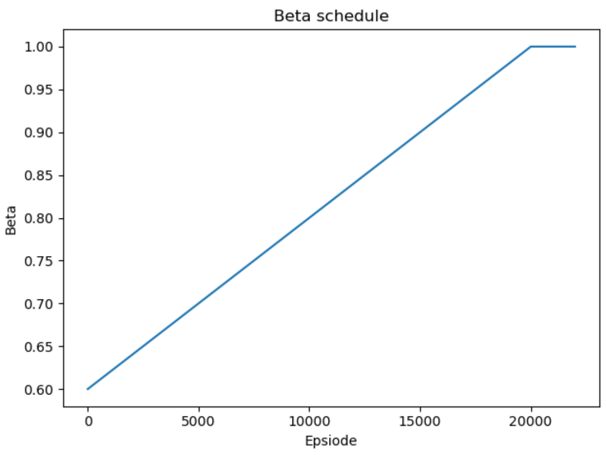


Figure 3: β schedule for training

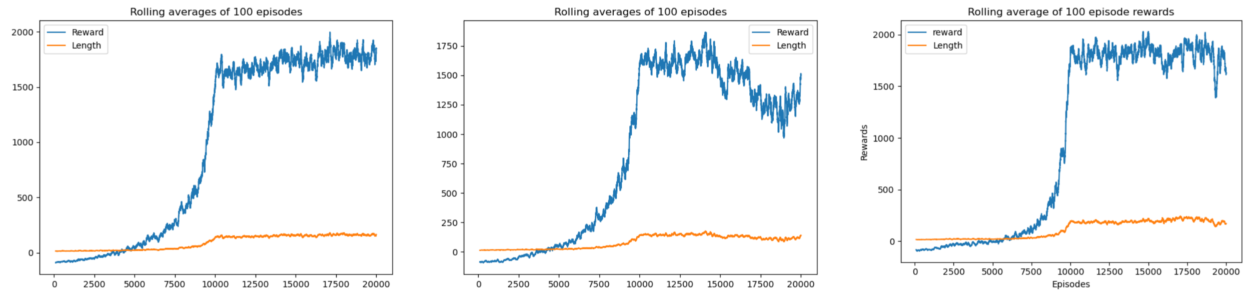
5.1 Simple

Using this state space models the problem as a Partially Observable Markov Decision Process since only a part of the entire state (the grid) is observed by the agent at every time step. This state space has two major consequences:

i. The model is not tied to any specific grid size, since we only use local information around the snake's head. This means that a model trained using a specific instance of the game can be run on any other instance, not necessarily of the same grid size.

ii. The disadvantage is that, since the agent cannot observe the entire grid, it is more likely to move into situations where it traps itself.

Figure 4 shows reward gained by the agent over a horizon of 20000 episodes by each of the three algorithms discussed in section 4. As expected, vanilla DQN performs worse than the other two. Double DQN with prioritized sampling performs equivalent to the tabular Q learning method, which is the best possible for this state space. For all three algorithms, the length of episodes increases after the reward increases. This makes perfect sense since this means that the snake is learning to eat the fruit rather than just learning to not collide, which is what we want.



(a) Tabular Q Learning (b) Vanilla DQN (c) Double DQN with Priority

Figure 4: Comparison of performance of the three algorithms using the simple state space.

The deep neural network architecture consisted of 3 fully connected layers of 16, 32, and 4 nodes, with the first 2 layers activated by the ReLu function. Each layer was initialized by He initialization He et al. [2015]. To train, we used the Adam optimizer with L2 regularization. We used a replay buffer of capacity $10,000$, a batch size of 32. We used $\alpha = 0.7$ as suggested in Schaul et al. [2015]. We copied weights from the training network to the target network every 4 steps.

A higher value of the discount factor γ makes it harder to learn, since the agent gives more and more weightage to future rewards. A comparison of reward curves for 3 values of the discount factor is given in figure 5. We use a value of $\gamma = 0.9$ for all our experiments since this gives the best performance empirically.

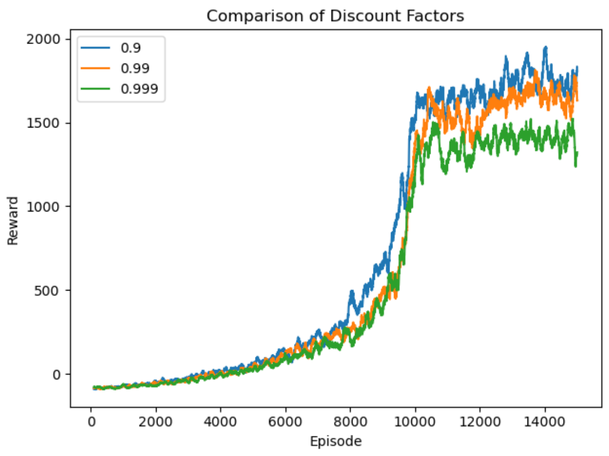
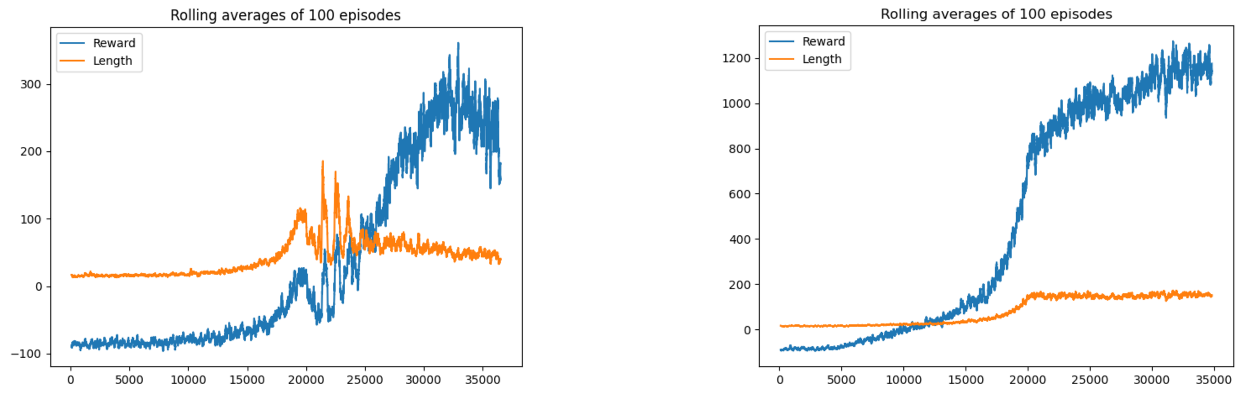


Figure 5: Comparison of reward curves for different values of the discount factor γ.

5.2 2D

On a grid of size just 10×10, running the tabular Q learning takes along time, since it required each state action pair to be visited multiple times to converge. However, this state space is large, since each of the 100 entries on the matrix can take 4 values, and there are 4 actions in the action space. This makes tabular Q learning impractical.



(a) Vanilla DQN

(b) Double DQN with Priority

Figure 6: Comparison of performance by algorithms using 2D state space.

Figure 6 shows reward gained by the vanilla DQN and the double DQN with prioritized sampling agents over a horizon of about 35000 episodes. Vanilla DQN struggles to learn anything useful, which is fixed by the enhancements that double DQN with priority brings in.

The input to the neural network is a 10×10 matrix. The first hidden layer convolves 16 5×5 filters with the input image and applies a ReLU nonlinearity. The second hidden layer convolves 32 3×3 filters again followed by a ReLu activation function. The final hidden layer is fully connected and consists of 256 rectifier units. The output layer is a fully connected linear layer with 4 outputs, one for each valid action. All layers are initialized with He initialization He et al. [2015]. To train, we used the Adam optimizer with L2 regularization. We used a replay buffer of capacity $50,000$, a batch size of 32. We used $\alpha = 0.7$ as suggested in Schaul et al. [2015]. We copied weights from the training network to the target network every 4 steps.

5.3 Framestack

Tabular Q learning with this state space suffers from exactly the same issues described above in section 5.2. Since there are 4 channels in each state (as we stack 4 frames) the state space is even larger. For this, we test only the double DQN with prioritized sampling, whose reward curve is given in figure 7.

The neural network is exactly the same as described in section 5.2 except that the input layer is a $4 \times 10 \times 10$ matrix since we have 4 channels. The training setup is also the same as described in section 5.2.

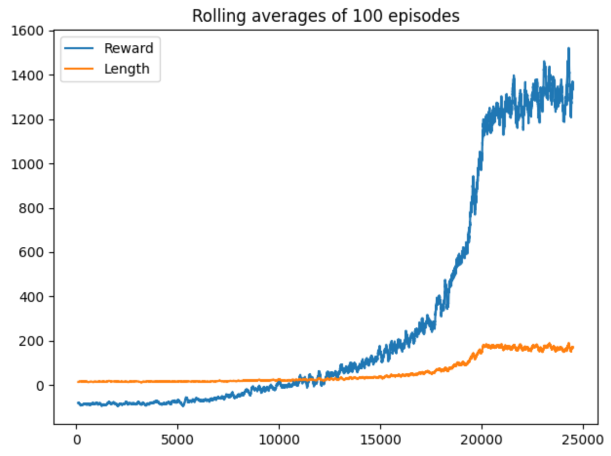


Figure 7: Reward curve for the double DQN algorithm with prioritized sampling for Framestack state space.

The additional information of direction encapsulated by each state enables the agent to achieve a higher reward than in the case for 2D state space.

6 Conclusion

In this report, we presented three algorithms to learn the snake game. We also tested these algorithms using three different states spaces, one of which models the game as a Partially Observable

Markov Decision Process. The enhancements done to improve the performance of vanialla DQN help the agent learn efficiently and effectively.

The agent learns to play the game using each of the three state spaces we tested on. In a small state space, Q learning performs quite effectively, but it is inefficient and impractical for large state spaces.

References

Norishige Chiba and Takao Nishizeki. The hamiltonian cycle problem is linear-time solvable for 4-connected planar graphs. *Journal of Algorithms*, 10(2):187–211, 1989. ISSN 0196-6774. doi:10.1016/0196-6774(89)90012-6.

Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL `https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL `http://arxiv.org/abs/1502.01852`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015. URL `http://arxiv.org/abs/1511.05952`.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. doi:10.1038/nature16961.

J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997. doi:10.1109/9.580874.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL `http://arxiv.org/abs/1509.06461`.

Giovanni Viglietta. Gaming is a hard job, but someone has to do it! In Evangelos Kranakis, Danny Krizanc, and Flaminia Luccio, editors, *Fun with Algorithms*, pages 357–367, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-30347-0_35.

Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. URL `http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf`.