Deep Q-Network: Multi-Agent Survival Strategy

Andy He
andyhe
andyhe@umich.edu

1. Introduction

My project dives into the implementation and analysis of the Deep Q-Learning Network (DQN), a reinforcement learning model renowned for its proficiency in mastering Atari games from raw pixel inputs. Atari Breakout serves as a benchmark in demonstrating how deep reinforcement learning models like DQNs can learn to control policies directly from raw sensory data inputs, as first introduced by Deepmind Technologies [1]. Building upon this foundation, I introduced another approach: deploying two independent DQN agents to engage in self-play and training against each other to enhance their strategies. This methodology aims to explore the dynamics of competitive learning and the emergence of sophisticated behaviors without external supervision.

I explored the dynamics of self-play in Deep Q-Learning Networks (DQNs), focusing on competitive strategy development through dual-agent interaction. Unlike traditional supervised learning, this approach fosters adapt-ability by challenging agents to evolve against an unpredictable opponent.

My contributions include the implementation of dual DQN agents, analysis of new strategies, and evaluation of competitive learning dynamics. This work provides insights into multi-agent reinforcement learning for applications such as game testing and strategic simulations.

2. Related Work

Reinforcement learning has been instrumental in advancing AI capabilities, particularly in complex games, as seen in DeepMind's AlphaGo, which utilized self-play to achieve superhuman performance in Go, and OpenAI's Five, which mastered Dota 2 through extensive self-play training [1]. However, these systems often rely on sophisticated architectures and significant computational resources. Further investigations, such as Pathway's work on AlphaXos, have implemented self-play mechanisms using DQNs in board games, highlighting the potential for DQNs to learn effective strategies without human intervention [2]. Furthermore, studies such as those of Huang and Zhu have applied deep multi-agent reinforcement learning to games such as Tic-Tac-Toe, where two agents learn by playing against each other, showcasing the applicability of DQNs in simple competitive scenarios [3]. Research has explored the application of deep Q-Networks in multi-agent environments. Tampuu et al. (2015) extended the DQN architecture to multi-agent settings by training two independent DQN agents to play against each other in the game of Pong [4]. Their study demonstrated the emergence of both competitive and collaborative behaviors, depending on the reward structures used.

These studies underscore the versatility of DQNs in multi-agent contexts and provide a foundation for exploration of self-play between DQN agents. I build on this foundation to analyze self-play dynamics and reward schemes.

3. Methods

3.1. Overview

The Deep Q-Network (DQN) is a reinforcement learning model designed to learn optimal policies for decision-making problems, such as playing video games or controlling robotic systems. It combines Q-learning, a value-based reinforcement learning algorithm, with deep neural networks to approximate the Q-value function, which represents the expected reward for taking an action in a given state. The model takes raw observations (e.g., images or states) as input and outputs Q-values for all possible actions. By selecting actions with the highest Q-values, the DQN learns to maximize cumulative rewards over time.

The model uses experience replay to store past experiences (state, action, reward, next state) in a buffer, randomly sampling them to break correlation in training data and improve stability. It also uses a target network to provide stable target values during training, reducing oscillations in Q-value updates. The convolutional layers in the DQN extract spatial features from the input (e.g., pixels), while the fully connected layers predict Q-values.

The study "Multiagent Cooperation and Competition with Deep Reinforcement Learning" by Tampuu et al. examined how two agents, controlled by independent Deep Q-Networks (DQNs), interact in Pong under different reward schemes [4]

- **Fully Competitive:** A zero-sum game where one agent's success directly penalizes the other (e.g., left player scores +1, right player scores -1).

	Left player scores	**Right player scores**
Left player reward	-1	1
Right player reward	1	-1

Table 1. Zero-Sum Strategy

- **Fully Cooperative:** Both agents lose a point whenever the ball leaves the boundaries of the game, encouraging them to keep the ball in play.

	Left player scores	**Right player scores**
Left player reward	-1	-1
Right player reward	-1	-1

Table 2. Cooperative Strategy

Although these strategies are conceptually interesting, I found them unsuitable. In the Fully Competitive scheme, rewards are delayed, leading to agents being rewarded for actions unrelated to the final outcome. In the Fully Cooperative scheme, agents are penalized for events beyond their control (e.g., one bot is punished for the other's mistake). To address these limitations, I implemented a ”survival strategy” reward scheme focused on individual accountability. Each agent is responsible for defending its own goal: if a bot allows a score, it is penalized (-1), while the opponent receives no penalty (0). In contrast, successful deflections are rewarded (+1). This ensures that rewards are directly tied to actions, promoting more fair and effective learning.

	Left player scores	**Right player scores**	**Left player hits ball**	**Right player hits ball**
Left player reward	0	-1	1	0
Right player reward	-1	0	0	1

Table 3. Survival Strategy

Therefore, I devised a ”survival strategy” reward scheme. Here, the focus is on individual accountability. Each agent is solely responsible for defending its goal. If the left slider agent allows the ball to score, it is penalized with a reward of -1, while the right slider agent receives no reward adjustment (reward = 0), as it has no control over the left slider agent's mistake. Moreover, if an agent successfully deflects the ball, it receives a reward (reward = 1). This approach ensures that training rewards are more directly tied to each agent's actions, fostering fairer and more effective learning for both agents.

3.2. DQN Architecture

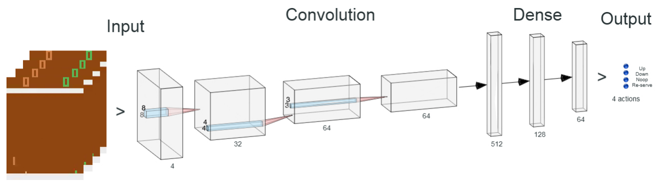
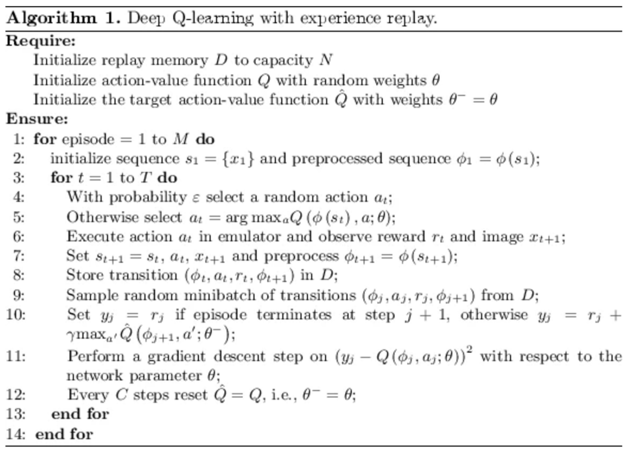


Figure 1. Neural Network Structure For DQN

3.3. DQN Pseudocode



```
Algorithm 1. Deep Q-learning with experience replay.
Require:
    Initialize replay memory $D$ to capacity $N$
    Initialize action-value function $Q$ with random weights $\theta$
    Initialize the target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
Ensure:
 1: for episode $= 1$ to $M$ do
 2:   initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
 3:   for $t = 1$ to $T$ do
 4:     With probability $\varepsilon$ select a random action $a_t$;
 5:     Otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$;
 6:     Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$;
 7:     Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$;
 8:     Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$;
 9:     Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$;
10:     Set $y_j = r_j$ if episode terminates at step $j + 1$, otherwise $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$;
11:     Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameter $\theta$;
12:     Every $C$ steps reset $\hat{Q} = Q$, i.e., $\theta^- = \theta$;
13:   end for
14: end for
```

Figure 2. DQN Algorithm Pseudocode [5]

4. Experiments

I implemented a custom DQN model with a unique reward system and compared its performance to the baseline in Tampuu et al. [4]. The study evaluated agent behavior using metrics like paddle hits per point, wall bounces per paddle bounce, and serving time per point. I adapted these metrics, focusing on paddle bounces per point and introducing cumulative points per episode to better capture learning progression under my ”survival strategy” reward system.

- **Paddle bounces per point vs. epoch:** Measures how many paddle hits occur before a point is scored.
- **Wall bounces per paddle bounce vs. epoch:** Tracks how frequently the ball hits walls relative to paddle bounces.

- **Serving time per point vs. epoch:** Monitors the time taken to serve the ball after a point is scored.

My approached differed in several ways:

- Instead of using "epochs" (250,000 steps) as the measurement unit, I opted for "episodes," where each game (first player to 21 points) is treated as one episode. This change makes agent progress easier to visualize and interpret.
- I excluded the "wall bounces per paddle bounce" metric, as it is less relevant in our "survival strategy" environment, where the primary goal is to prevent goals rather than focus on wall-based tactics.
- I counted frames per point instead of time, although this does not affect the overall trends in the graph since time can still be deduced based on the frame count.

To enhance my analysis, I introduced a new metric: cumulative points per episode, which captures whether both agents improve their performance over time. This metric provides a clearer indication of the agents' learning progression as they adapt to survive and outperform each other.

4.1. Results

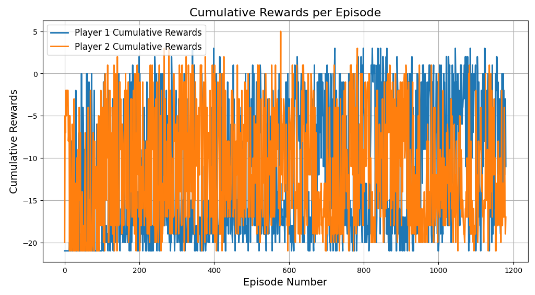


Figure 3. Cumulative Rewards Per Episode

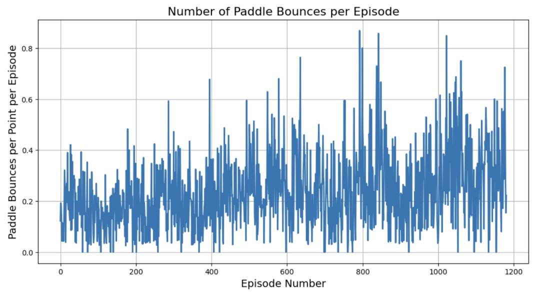


Figure 4. Number of Paddle Bounces Per Episode

Due to budget and time constraints, I trained the model over 1,180 episodes, whereas Tampuu et al. 's study [7] trained their agents for over 50 epochs. Consequently, I was

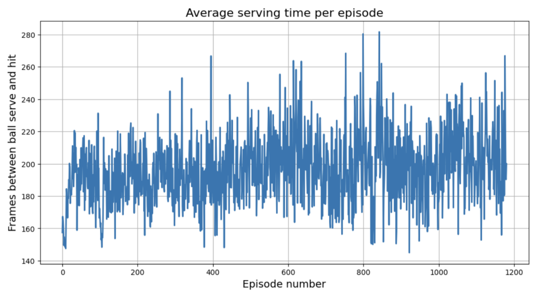


Figure 5. Average Serving Time Per Episode

unable to fully replicate the study's results. However, based on their findings—where minimal improvement in "paddle bounces per point" occurred until around the 48th epoch—I hypothesize that with greater computational resources, the agents could achieve similar outcomes. Scaling up training time would likely enable the agents to develop more advanced strategies and converge to behaviors observed in the baseline study.

4.2. Hyperparameters

To train the model, I used an epsilon decay of 0.92 with a minimum epsilon of 0.10. The epsilon value controls how many of the actions taken during a training session are randomly selected rather than provided by the model's output. This is used to allow the model to train on input it would normally not generate itself. Having an epsilon value too high can result in the model not being able to use learned behaviors, making it less efficient or impossible to train. Setting an epsilon value too low can result in the model sticking to a learned behavior rather than exploring more optimal behaviors. The model also used an Adam optimizer for both the target and online models with a learning rate of 0.00025. The gamma value, or discount rate, that I used to train the model was 0.99, which is the multiplier given to past actions during training.

5. Conclusion

This project demonstrated the potential of self-play in enhancing DQN agents' strategic capabilities. Due to time constraints, I can only observe the agents' fluctuating performance as it continues to learn and train. Consequently, I was unable to fully replicate the study's results. However, based on their findings—where minimal improvement in "paddle bounces per point" occurred until around the 48th epoch in their graph [4], I hypothesize that with greater computational resources, the agents could achieve similar outcomes. Scaling up training time would likely enable the agents to develop more advanced strategies and converge to behaviors observed in the baseline study. However, achieving benchmarks set by studies with longer training times, such as DeepMind's 38-day experiments [6], underscores the importance of extended training.

My ”survival strategy” reward scheme addressed fairness and individual responsibility, offering a viable alternative to fully competitive and cooperative schemes. Future work could explore advanced strategies, training optimizations, and additional techniques like frame skipping or dueling networks.

This study highlights DQN agents' utility in competitive multi-agent environments and the need for further research to maximize their potential.

References

[1] Christopher Berner, Greg Brockman, Brooke Chan, et al. Dota 2 with large scale deep reinforcement learning, 2019.

[2] Robin Ranjit Singh Chauhan. Alphaxos: A multi-agent reinforcement learning framework for board games, 2022.

[3] Zhi-Wen Huang and Wei-Jin Zhu. Tic-tac-toe with deep multi-agent reinforcement learning. *University at Buffalo RL Workshop Proceedings*, 2019.

[4] Ardi Tampuu, Tambet Matiisen, et al. Multiagent cooperation and competition with deep reinforcement learning. *arXiv preprint arXiv:1511.08779*, 2015.

[5] Fuxiao Tan, Pengfei Yan, and Xinping Guan. Deep reinforcement learning: From q-learning to deep q-learning. In Derong Liu, Shengli Xie, Yuanqing Li, Dongbin Zhao, and El-Sayed M. El-Alfy, editors, *Neural Information Processing*, pages 475–483, Cham, 2017. Springer International Publishing.

[6] Keras Team. Deep q-learning (dqn) example for playing breakout. `https://keras.io/examples/rl/deep_q_network_breakout/`. Accessed: 2024-12-11.