CS238 Final Project: Q-learning with Customized State Space Reduction on Atari Breakout Game

Ya-Chi Chu
Department of Mathematics
ycchu97@stanford.edu

Nicolas Friley
Department of Electrical Engineering
nfriley@stanford.edu

Abstract

This project focuses on training a Q-learning agent to play the Atari game Breakout. The problem is reformulated as a partially observable Markov decision process (POMDP). Efforts were made to reduce the dimension of the state space, resulting in a more compact state representation. The agent was trained using Q-learning with an ε-greedy exploration strategy. The trained agent outperformed a baseline random agent and demonstrated improved gameplay in Breakout.

1 Introduction

Atari 2600 games is a challenging testbed for reinforcement learning algorithms. In this project, we focus on one specific game called Breakout among all the 2600 games. In Atari Breakout game, the player controls the position of a paddle at the bottom of the screen to bounce a ball upwards and break a brick wall. Bricks of different colors have varying point values. If the ball falls below the paddle, the number of lives decreases. The game ends when there are no more bricks left or no lives remaining. The game involves the uncertainty of the ball's position and its direction/speed after hitting the paddle. Players sequentially make a decision on how to move the paddle based on the current configuration of the game.

Based on the Atari Breakout game environment provided by OpenAI Gym, we trained a Q-learning agent to play the game, which performs significantly better than the baseline random agent. The goal of the agent is to interact with the emulator by selecting actions in a way that maximizes future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted return as $R = \sum_{t=0}^{T} \gamma^t r_t$, where T is the time-step at which the game terminates.

2 Related Works

The Atari 2600 emulator, initially explored by Bellemare et al. [2013], laid the foundation for reinforcement learning. They applied standard algorithms with linear function approximation and generic visual features, with subsequent advances like random feature projection Bellemare et al. [2012] and integration of HyperNEAT evolutionary architecture Hausknecht et al. [2014]. Post-2013, there was a notable shift to deep reinforcement learning for training Atari 2600 game agents, marked by the introduction of the Deep Q-network (DQN) by Mnih et al. [2013]. This transition was followed by works like DQN with offline Monte-Carlo Tree Search Guo et al. [2014], Double Deep Q-network (DDQN) Van Hasselt et al. [2016], Dueling DQN de Freitas [2015], and NoisyNet Fortunato et al. [2017], collectively enhancing agent training.

Recent efforts in deep reinforcement learning focus on reducing training instances and associated time. Ho et al. [2019] presented a system rapidly learning causal rules in an Atari game environment, achieving human-like performance in score and time. Wu et al. [2023] introduced the Read and

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

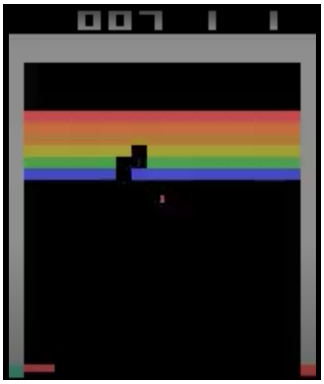

Figure 1: A screenshot of Atari Breakout game.

Reward framework, leveraging human-written instruction manuals for policy learning in Atari game agents. These advances reflect ongoing endeavors to boost efficiency and efficacy in deep reinforcement learning for Atari games.

3 Problem Formulation

OpenAI Gym Brockman et al. [2016] provides environments for the Atari Breakout game. In our project, we use the environment `Breakout-v5`. Every time after the agent takes an action, the environment provides an RGB image of the screenshot of the game (see Figure 1) and the number of lives remaining. Instead of taking the RGB images as the states and training an agent based on the this extremely high dimensional state space, we treat the problem as a POMDP with full observability. That is, we assume that the state can be deterministically extracted from the observation. We view RGB images as observations, design the state space to reduce dimension, and extract the state from each observation (RGB image) at every step. Specifically, we formulate the problem as a POMDP $(\mathcal{S}, \mathcal{A}, \mathcal{O}, T, R, O, \gamma)$ as follows:

- **Observation Space $\mathcal{O}$.** Each observation is an RGB image represented as a 3D array with dimensions $210 \times 160 \times 3$ (width $\times$ height $\times$ number of color channels), the value of which lies between 0 and 255.
- **Action Space $\mathcal{A}$.** The action space $\mathcal{A}$ consists of four actions: {'NOOP', 'FIRE', 'RIGHT', 'LEFT'}, where 'NOOP' means no move, 'FIRE' means restarting the game after dropping previous ball, and 'RIGHT'/'LEFT' means moving paddle to the right/left.
- **State Space $\mathcal{S}$.** We represent the state s as a composite information comprising the presence of each brick, the position of the ball, and the position of the paddle.
- **Transition Model T.** When action a is taken at state s, the transition model $T(s' \mid s, a)$ describes the probability of transitioning to a new state s', consisting of new ball location, new paddle location, and the updated presence of the ball.
- **Reward Function R.** The reward is the points that the agent gets when the ball hits the brick(s). The color of a brick determines the value of the reward: red and orange for 7 points; brown and yellow for 4 points; and green and blue for 1 points.
- **Observation Model O.** When the agent takes action a at state s, the observation model determines the RGB screenshot of the game. We assume the observation model is deterministic. In other words, given a state s and an action a, out agent will observe a unique observation without the noise. Hence, we may deterministically extract the state from our observation and do not need to update the beliefs.
- **Discount Factor γ.** The discount factor is fixed to be $\gamma = 0.95$ through the whole project.

4 Approach

4.1 Explored Strategies on Larger State Space

Before attempting dimension reduction on the state space, we directly trained a Q-learning agent on the RGB images. However, due to the Q-function's Python dictionary structure, where keys are states s and values are 1D arrays of length $|\mathcal{A}| = 4$ containing Q-values for the state-action pair (s, a), hashing a 3D array of shape $(210, 160, 3)$ required flattening in each iteration. This process significantly slowed down training, with 500 episodes taking over 20 minutes, resulting in an agent that failed to move the paddle before exhausting 5 lives.

A more compact yet less insightful state space attempted was the 128-Byte RAM observation provided by our Breakout game environment. Each observation was represented as a one-dimensional array of 128 bytes, offering a compressed representation of the original game screen. We take this observation as the state directly. Despite faster training (over 5000 episodes with decaying ε from 1.0 to 0.1), the agent performed poorly. Upon restarting the game after losing a life, the paddle moved randomly for several time steps, eventually ceasing without hitting the ball within 5 lives.

Concerning the poor performance of Q-learning agent on large or meaningless state space, we decide make efforts on the state space reduction. See more details in section 4.2.

4.2 Dimension Reduction for State Space

Different methods must can used to handle the very large dimension of the original state space. One of them is using Deep Q-Learning on the full observation space. Because we wanted to constraint ourselves to regular reinforcement learning methods, we decided to develop a mapping between the observation space and a much-reduced state space that would work well with Q-learning.

In the image returned as an observation, we decided to keep track of the position of the ball, the position of the paddle, and a survey of the brick wall.

1. We encoded the position of the paddle with a simple integer representing the middle pixel of the paddle within the width of the image. For example, the width of the image is 160. If the paddle was only 1 pixel wide, the value would be an integer between 0 and 160. But because the paddle had a width of 16 pixels, the value of the middle pixel (chosen to be the rightmost middle pixel) was between 9 and 153.

2. To encode the situation of the brick wall, we checked the color value of one pixel for each of the 108 bricks. If the color value was 0, then the screen was black where the brick should be, indicating a missing brick. Our wall vector would then get a 0 for this brick. If the brick was still present, the wall vector would get a 1 for this brick.

3. To encode the position of the ball, we checked for an object of 2 by 4 pixels of the particular color value the ball had. Because the paddle as well as the top brick wall had the same color value as the ball, the code would sometimes mistake a chunk of them for the ball. To keep the code relatively simple and prevent errors in ball detection, we decided to only check for the ball location in the segment of the screen below the brick wall and above the paddle line. This decision was motivated by the fact that the paddle would only move based on the ball movement in this segment, and would not necessarily move based on the movement on the ball inside the brick wall. Additionally, if the ball went below the paddle line, a life would be lost anyway, and the movement of the paddle wouldn't matter anymore. If the ball was not located, which means it was either above the lower brick line, or below the paddle line, the position of the ball would be encoded as (-1, -1). The rest of the time, the pixel location of the ball was returned as (x, y).

Our state space ended up being a tuple of 111 integer values.

4.3 Baseline: Random Agent

At each time step, the random agent selects a random action a from the action space $\mathcal{A}$. In other words, the decision made by the agent is independent of the observation from the environment.

4.4 Q-Learning with ε-greedy Exploration

The transition model in Atari Breakout is not accessible. Moreover, even after the dimension reduction of the observation space, the dimension of the state space is still very large. Therefore, instead of using the model-based method which might yield low-quality estimation of transition model, we decide to use Q-learning, a model-free reinforcement learning method.

Q-learning uses an incremental update of the mean to estimate the action value function $Q(s, a)$. At each time step with state-action pair (s, a), an update to the action value function is performed as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q\left(s', a'\right) - Q(s,a) \right),$$

where s' is the next state, α is hte learning rate, r is the reward, and γ is the discount factor. Note that each update of Q-learning takes time $O(|\mathcal{A}|)$, so the algorithm can take the advantage of our small action space $|\mathcal{A}| = 4$.

In our implementation, the Q-values are stored in a dictionary. The keys are all the states that were encountered in the form of 111-element tuples, and the respective values are the q-values associated to each of the 4 possible actions for this state. When a new state is encountered, a new key is added to the dictionary and the q-values are initialized to zero for all actions.

4.4.1 Exploration Strategy

At each time step, the ε-greedy exploration strategy takes a random action with probability ε and takes a greedy action $a = \arg\max_{a'} Q(s, a')$ based on current action value function with probability $1 - \varepsilon$. In other words, the larger the value of ε is, the more the agent explores. During our training process. we incrementally decreased the value of ε. The agent is trained for a total number of 300,000 episodes. Starting with $\varepsilon = 1$, we incrementally decreased the value of ε by 0.1 at every $30,000$ episodes but kept the value of ε not less than 0.1.

5 Results and Analysis

5.1 Averaged Rewards during Training Process

We recorded the rewards of our agent during 300,000 training episodes and represented the average of every 1000 rewards in figure 2. The plot shows a clear increase in average reward at each 30,000 threshold where the epsilon-greedy parameter has been incrementally decreased as discussed in 4.3.1. However, the overall progress seems to head towards a plateau at a reward of around 3. This means that with the current definition of our state space, the agent might not be able to learn past a certain reward level. One hypothesis concerning the limits of our state space definition is the fact that the Cartesian coordinates of the ball do not capture the sequential position of the ball. This means that for any position of the ball on the screen, it could actually be heading up or heading down, left or right. But the agent does not receive directional information. One way to implement this would be to save the current Cartesian coordinates of the ball in a temporary variable, and add this variable to the next state, next to the new Cartesian coordinates of the ball. This way, we would have $x_{t-1}, y_{t-1}, x_t, y_t$ in each state instead of simply x_t, y_t.

5.2 Completely Greedy Agent v.s. Not so Greedy Agent

Figure 3 shows the boxplot for the rewards obtained by our agent when playing 1000 games for different values of the epsilon-greedy parameter. With an epsilon of 0.6, the boxplot shows several high-reward outlier groups during 1,000 games played. This hints that our agent would still strongly benefit from a high exploration, even after having been trained for 300,000 episodes. Additionally, the distributions for rewards over 1,000 games is very similar no matter the level of greediness. This suggests that the agent's action is not influenced by the Q-values for a large number of states. This happens when the agent encounters a state that wasn't previously in the dictionary, since the q-values for all actions get initialized with zeros. With a large enough number of training episodes, the distribution of rewards for 1000 games should be one line located at the maximum possible score achievable in the Breakout game. This would only be possible if the Q-value dictionary contained a key and a set of trained q-values associated to actions, for almost every possible state in the state space.

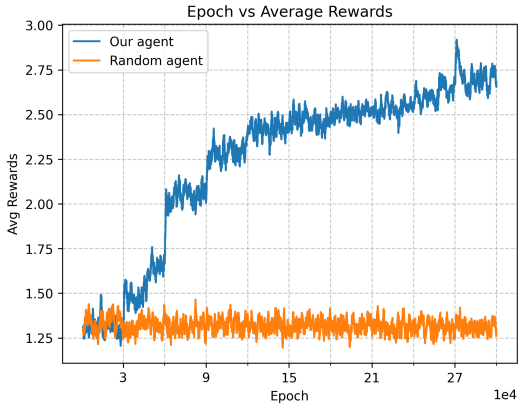

Figure 2: Averaged rewards of every 1000 episodes during training process.

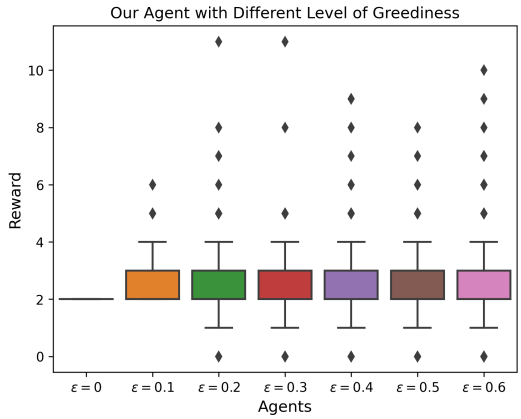

Figure 3: Final rewards of 1000 games with different level of greediness.

However, our Q-value dictionary possibly only contains keys for an extremely small percentage of the state space due to not enough training. This results in our agent only being able to play correctly for the first rebound of the ball.

Currently, the highest rewards obtained follow the same pattern:

1. The paddle moves to a corner and intercepts the ball, successfully bouncing it towards the bricks.
2. The paddle doesn't move and the ball drops, leading to a loss of one life.
3. The ball is re-inserted, and happens to be shot in the exact same trajectory as the first ball.
4. It bounces again on the paddle, hits a brick and drops, leading to a loss of one life, Etc.

As a result, at least one brick is struck for every life. Unfortunately, the game ends extremely fast, and the max score is still capped at 13, which remains extremely low overall. This means that with 300,000 episodes of training, the agent has only learned to optimize the first action to make, but is incapable of following up with more actions to keep one ball in the game for longer. One possible explanation is the fact that the agent tries to maximize the reward over all 5 lives, instead of trying to maximize the reward for each life. When the agent learns that by not moving, it can sometimes secure a higher reward for each new ball, it doesn't try different actions to maximize rewards by saving the ball. In a sense, the agent is too greedy. This could be addressed by changing the definition of the Terminated boolean variable, so that the game terminates once the number of remaining lives changes.

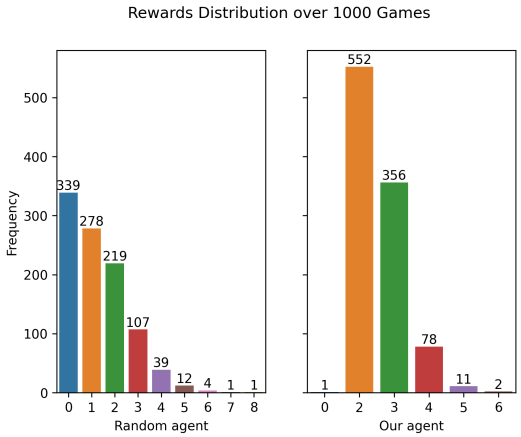

Figure 4: Distribution of rewards of random agent and our agent.

5.3 Reward Distribution over 1000 Games

Figure 4 shows the distribution of final points got by random agent and our agent with greedy level $\varepsilon = 0.1$. We can clearly see a distribution shift toward the positive direction after training. Most of the time, the random agent got 0 points after losing 5 lives, while our can almost always get more than 2 points through the whole game. Half of the time our agent can only get 2 points, but for another quarter of time, it can manage to hit the ball one more time to get an additional 1 point.

6 Conclusion & Future Works

After analyzing our results, we have identified several improvements that can enhance the training of the Q-learning agent. Firstly, we propose incorporating the direction of the ball into the state space. The second improvement involves training the agent on a single life only, encouraging the model to maximize rewards for each individual life. This approach helps mitigate the model's inclination to prioritize immediate rewards from firing the next ball, rather than prolonging the current ball's presence in the game.

Alternatively, a complex reward function could be devised to address this issue. This reward function could inflate the incentive for saving the ball over obtaining a reward with the next ball. Additionally, a "penalty" for losing a life could progressively increase as the number of lives approaches zero to emulate the mounting pressure experienced by human players as their remaining lives dwindle.

Some rewards could also be added every time the direction of the ball shifts from downwards to upwards, indicating that the paddle successfully bounced the ball upwards, and is likely to hit bricks as a result. This bouncing reward could be very high as the brick wall is mostly intact, to express the high likelihood of transforming a successful bounce into a broken brick, and could decay as the brick wall gets broken, so that the model can learn to prioritize hitting particular bricks over simply bouncing the ball randomly.

If these modifications result in a reward plot that demonstrates a more pronounced linear increase compared to our current results, it is highly likely that increasing the number of training episodes would yield a significantly stronger Q-learning agent. To achieve consistent scores exceeding 10 points, we estimate that over 1 million training episodes may be necessary.

Considering that the maximum achievable score in this game is 432, training an optimally performing agent capable of reaching this score would likely require a much larger number of episodes. It is challenging to directly compare the training time and performance of a Q-learning agent with those of a Deep Q-Network (DQN). However, it is plausible that even if the Q-learning agent can attain optimal gameplay, the duration of training would dramatically surpass that reported in the DQN literature.

7 Contributions

Both team members worked together on formulating the problem, discussing the approaches, making adaptations to the algorithms, running the code, analyzing results, and contributing to the write-up of this report.

References

Marc Bellemare, Joel Veness, and Michael Bowling. Sketch-based linear value function approximation. *Advances in Neural Information Processing Systems*, 25, 2012.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47: 253–279, 2013.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Nando de Freitas. Dueling network architectures for deep reinforcement learning. Technical report, 2015.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. *Advances in neural information processing systems*, 27, 2014.

Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

Seng-Beng Ho, Xiwen Yang, Therese Quieta, Gangeshwar Krishnamurthy, and Fiona Liausvia. On human-like performance artificial intelligence: A demonstration using an atari game. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 9–12, 2019.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Yue Wu, Yewen Fan, Paul Pu Liang, Amos Azaria, Yuanzhi Li, and Tom M Mitchell. Read and reap the rewards: Learning to play atari with the help of instruction manuals. *arXiv preprint arXiv:2302.04449*, 2023.