Comparing Reinforcement Learning Algorithms in Ms. Pac-Man

Ali Jameel
101185339
alijameel@cmail.carleton.ca

Alexander Gruescu
101202529
alexandergruescu@cmail.carleton.ca

Dominic Lau
101240377
dominiclau@cmail.carleton.ca

Sara Francis
101234656
sarafrancis@cmail.carleton.ca

Abstract

There are many reinforcement learning algorithms that have been successfully applied to video games. The complex environments provided by video games offer a unique space for implementing and testing reinforcement learning algorithms. In this paper, we analyze different reinforcement learning algorithms in the Atari 2600 video game, Ms. Pac-Man, and seek to compare and contrast the performance of these algorithms.

1 Introduction

Video games have long been a typical application for reinforcement learning. The interactive environments they provide offer a virtual playground for agents to interact with. The wide variety of video games likewise allows for a wide variety of reinforcement learning-based approaches to be tested. In doing so, the complexity of video games presents an opportunity to analyze various different reinforcement learning algorithms and how they fair in nuanced environments.

In this report, we concern ourselves with the Atari 2600 video game, Ms. Pac-Man. In the Atari 2600 iteration of the game, Ms. Pac-Man involves a player-controlled agent in a maze-like environment (see Figure 1). The player can move within the maze by moving right, left, up, or down. The goal of the player is to eat all the pellets located on the maze whilst avoiding the colored ghosts, who will chase down the player. Collecting all the pellets will advance the player to the next level, with each level increasing in difficulty. The player's high score is determined by the number of pellets consumed, consuming ghosts when they are blue (which happens when a large pellet is eaten), as well as cherry and banana items which may spawn. The player's lives are displayed at the bottom left corner of the screen. Touching a ghost (unless they are blue) will cause the player to lose a life, if a player loses all their lives, the game is over. To complicate matters, the ghosts which pursue the player act upon unique decision policies, meaning that the player must react in real-time to the decisions of the ghosts.

To perform effectively, the player must balance between navigating the maze, maximizing their high score, and avoiding the pursuing ghosts. The combination of all these elements makes this game ideal for analyzing RL methods.

It is for this reason that we analyze the Ms. Pac-Man game with different RL algorithms. Although various papers have been published about specific approaches, we seek to compare them and find the strengths, weaknesses, and attributes of each approach. In doing so, we hope to unveil the viability of numerous RL approaches in wake of complex interactive environments.

Our goal will be to examine four main algorithms: SARSA, Q-Learning, Deep Q-Learning, and Actor Critic. Each of these algorithms provide their own challenges and are ideal in their own way. We aim to explore and differentiate the results between these algorithms, as well as walk through the thought process and decision making for the algorithms.

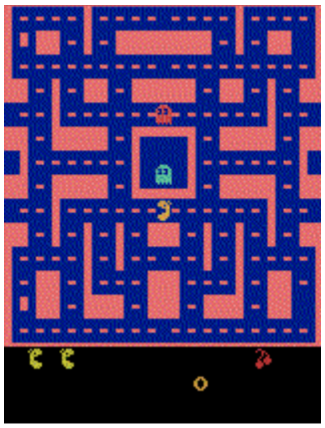

Figure 1: Atari 2600 Ms. Pac-Man game

2 Approaches

2.1 Prior Work

There exist many studies that have tested and analyzed reinforcement learning algorithms in Atari 2600 games from the Arcade Learning Environment. Here are some of the prior works:

Deep Q-learning:

One of the algorithms we were interested in is Deep Q Learning. A paper called “Playing Atari with Deep Reinforcement Learning” [1] introduced a way to use convolutional neural networks with Q Learning to predict the expected value of each action by taking raw pixels as inputs. First, they applied preprocessing on the pixels of the frame by applying a gray filter on the image and then they down-sampled and resized it to a 84 x 84 frame. Then they took the pixels and put them into the convolutional neural network to estimate the Q-values. Additionally, it also uses a technique called experience replay to store transitions (which consists of the current state, action, reward, next state, and terminated) at each time step to a data set, and then randomly samples a small batch of stored experiences to the Q-Learning updates.

Since the authors in that paper only tested their methods in seven Atari 2600 games (Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, and Space Invaders), we were curious as to how it would perform in other Atari 2600 games from the Arcade Learning Environment games such as Ms. Pac-Man.

Deep Reinforcement Learning with Double Q-Learning:

Another paper we found called "Deep Reinforcement Learning with Double Q-Learning"[2] showed massive improvement in some Atari 2600 games from the Arcade Learning Environment compared to the original Deep Q Learning paper [1]. In this paper, it explained how Q-learning algorithm is known to overestimate action values under certain conditions and proposed a solution to reduce this overestimation. Additionally, it showed how Double DQN, when tuned, can perform really well against regular DQN.

Unfortunately, when we tried to use Double Q-Learning in our Deep Q-Learning algorithm for Mrs. Pac-Man, it yielded unsatisfactory results, so it was not included.

Fuzzy Q-learning:

Finally, another paper we found interesting was about Fuzzy Q-Learning. Due to the non-deterministic nature of the environment, fuzzy classes can be applied to the game for the agent to make real-time decisions based on several contributors, such as the distance to the nearest pill or ghosts [3].

2.2 Game Model

To compare our algorithms, we used Ms. Pac-Man from the Arcade Learning Environment. There are 3 different ways to initialize the Ms. Pac-Man gym environment observation space: RGB, grayscale and RAM. The RGB initialization means there are 210 * 160 * 3 = 100800 features of the state because the board is 210 * 160 pixels and each pixel has 3 values, one value for each of the colour channels. The grayscale initialization means there are 210 * 160 = 33600 features of the state since each pixel is a feature and the board is 210 * 160. Finally, RAM initialization has only 128 features where each feature is a byte of data representing the environment.

There are 9 actions available to the agent as per the Gym environment: {NOOP, UP, RIGHT, LEFT, DOWN, UPRIGHT, UPLEFT, DOWNRIGHT, DOWNLEFT}. However, only 4 actions are actually feasible in the game itself: {UP, RIGHT, LEFT, DOWN}.

2.3 SARSA and Q-Learning (Linear Function Approximation)

We found that the RAM representation worked best for the SARSA and Q-Learning algorithms because there were far fewer values to multiply when estimating the Q values. This meant that training went much faster and the agent could play about 5 games a second.

For SARSA with linear function approximation, the algorithm works by first initializing the action parameters w to a 128 (state features) x 9 (actions) matrix of random values. The first action is chosen based on an ϵ-greedy algorithm where there is a 10% chance of choosing a random action and the rest of the time the agent chooses the action with the highest estimated q value (calculated by multiplying the 128 state features by w to get the 9 action values). The second action is chosen in the same way, and then the values in w for the first action are updated based on the step size, reward, discount factor and second action value. This continues until an action leads to a terminal state, at which point w is updated but the value of the second action value is 0, rather than whatever the function approximation with w would predict. Episodes are played in this way until training is complete.

Q-learning with linear function approximation is very similar to SARSA, the only difference being that the Q-learning algorithm updates the action parameters w based on the maximum estimated q value for the next state, but doesn't necessarily choose this action in the next time step, it uses an ϵ-greedy approach.

2.4 Actor-Critic

For the Actor-Critic implementation with linear function approximation, the algorithm works by first initializing the actor weights as a 9 x 128 (state features) matrix, and the critic weights as a 128 (state features)-length vector. The agent then begins each episode by selecting an initial action using the ϵ-greedy algorithm where there is a 10% chance of choosing a random action and the rest of the time the agent chooses the action with the highest probability, calculated by applying a softmax function to the actor's weight matrix. After each step, the agent calculates the value of the current state, and uses the reward, discount factor, and value of the next state to determine the TD error. The critic weights are updated to minimize the TD error, and the actor weights are updated to emphasize actions with higher rewards. The total reward is recorded for each episode, and the training continues until the decided amount of episodes are complete.

2.5 Deep Q-Learning

For the Deep Q-Learning algorithm, we found that using the grayscale representation works the best because the colors of the objects in the game were not critical in the agent's ability to learn the optimal strategies. By converting it to grayscale, it focuses on the more relevant aspects of the game such as the location of the ghosts and pellets and reduces the time and space complexities. Afterwards, we preprocessed the frames by resizing it to a smaller 110 x 84 frame (similar to what was in the paper) from the original 210 x 160 frame and dividing the pixels by 255. From there, we sent the pixels of the frames into a convolutional neural network with 3 convolution layers (with different strides) and 3 fully connected layers to predict the expected value of each action. Additionally, at each time step, we stored the experiences (transitions) into the "replay memory" and we would occasionally sample 32 batches of previously stored transitions to the Q-Learning updates. Finally we also used a target neural network which predicts the estimated Q-values.

Initially, for our implementation of the Deep Q-Learning algorithm, we used a convolutional neural network with 4 convolutional layers (default stride of 1), a pooling layer, and 3 fully connected layers. However, we noticed that when executing the algorithm, it took a very long time to finish. We eventually came across a poster called “Atari Breakout With Neural Networks” which had a convolutional neural network with 3 convolutional layers (each with different strides) and 2 fully connected layers [4]. The poster gave a nice analysis on the performance of their Deep Q-Learning algorithm using different amounts of layers and talked about their architecture such as their optimizer (Adam) and loss function (RMS).

Afterwards, based on the paper, we modified our convolutional neural network to have 3 convolutional layers instead of 4, added strides to them, and removed the pooling layer.

Convolution Layers				
Layer	In Channels	Out Channels	Kernel Size	Stride
1	1	16	3	4
2	16	32	3	2
3	32	64	3	1
Fully-connected Layers				
Layer	In Features	Out Features		
1	5362	128		
2	128	64		
3	64	9		

Table 1: 3 Convolutional layers (with different strides) and 3 fully connected layers

3 Empirical Studies

3.1 SARSA

Our experiments indicated that the agent performs best with a small learning rate ($\alpha = 0.001$) and a large discount factor ($\gamma = 0.99$). The agent tends to choose the action move right, receive a reward, move right again, receive a reward, etc., continuing to move right until the game has terminated when the ghosts have captured Pac-Man. However, when it reaches a wall it would continue to choose to move right, likely because it had just received a fair amount of rewards for that action and the values for the state hadn’t changed enough for the q value estimation to choose a different action. Using a small learning rate so the agent didn’t adjust the action parameters in w too much after the 5 or 6 successful move-right actions seemed to be the best way to combat this issue. Additionally, using a large discount factor (gamma) means that the agent prioritizes long-term rewards over immediate/short-term rewards, hopefully getting it to choose an action that results in it choosing a different action, so it doesn’t continue to run into the wall.

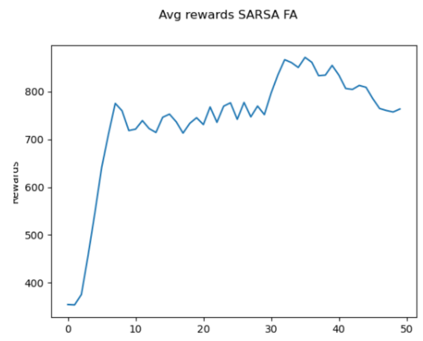

Figure 2: Rewards over 50,000 games of Ms. Pac-Man (averaged over every 1,000 games), using the SARSA function approximation algorithm

As seen in Figure 2, the agent learns well over the first 10,000 games and achieves about 750 points per game on average. However, after 10,000 games there is little improvement and, aside from a slight increase after 30,000 games, the agent does not seem to be improving its strategy. It is possible to do much better than 750 points in the game, but perhaps this is the best the agent can do with a relatively simple algorithm and only 128 state features.

3.2 Q-Learning

Since the Q-learning algorithm is very similar to the SARSA algorithm our experiments yielded similar results. The agent performs best with a small learning rate ($\alpha = 0.001$) and a large discount factor ($\gamma = 0.99$) so that the parameters in w are updated too quickly and the agent prioritizes long-term rewards over immediate rewards.

As seen in Figure 3, the agent learns well over the first 10,000 games and reaches about 725 rewards per game. However, after 10,000 games the amount of rewards remains relatively consistent, suggesting that the agent is not improving its strategy. Additionally, the extra data points from averaging every 500 games (rather than every 1000 like SARSA) show a variation in the score, from about 650 to 750 points. This is a relatively large variation, especially considering it's an average, and it suggests that the agent still isn't very consistent with its strategy. Furthermore, it is possible to do much better than 725 points in the game, but perhaps this is the best the agent can do with a relatively simple algorithm and only 128 state features.

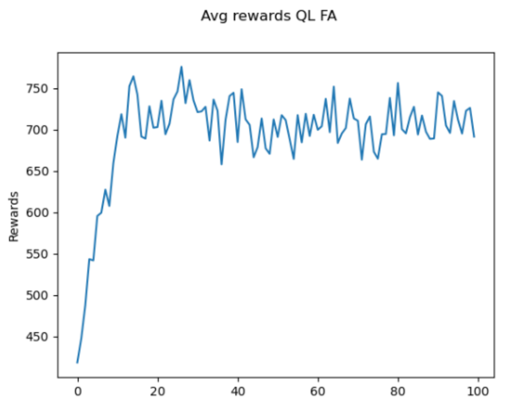

Figure 3: Rewards over 50,000 games of Ms. Pac-Man (averaged over every 500 games) using the Q-learning function approximation algorithm

3.3 Actor-Critic

Similarly to the SARSA and Q-Learning algorithm, we also used a high discount factor of $\gamma = 0.99$ so that the Pac-Man can focus on long term rewards over immediate/short term rewards. Additionally, we found that the optimal actor and critic learning rate is $\alpha = 0.0005$ for the actor learning rate and $\alpha = 0.001$ for the critic learning rate.

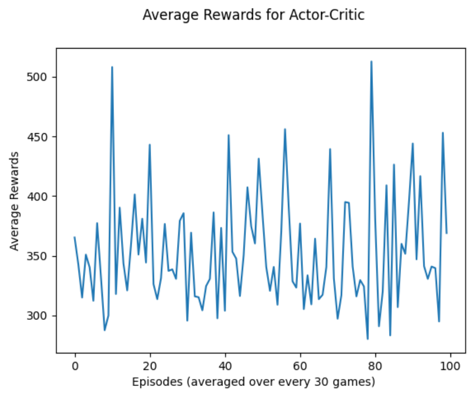

Figure 4: Rewards over 3000 games of Ms. Pac-Man (averaged over every 30 games) using the Actor-Critic algorithm

When we look at the graph in Figure 4, we can see that the Actor-Critic algorithm on average scores 350 points per game. This performance is only slightly better than that of a Pac-Man agent taking random actions, which typically scores around 300 points per game.

Additionally, when comparing the Actor-Critic algorithm to the SARSA and Q-Learning algorithm, we can see that it doesn't perform better as it experience more episodes. It flattens out at around 350 points per game vs. around 750 points per game for the SARSA algorithm and 700 for the Q-Learning algorithm.

3.4 Deep Q-Learning

For our Deep Q-Learning algorithm implementation, we learned it through reading the PyTorch Deep Q-Learning tutorial/documentation [5] as well as a YouTube series that teaches Deep Q-Learning for Flappy Bird [6].

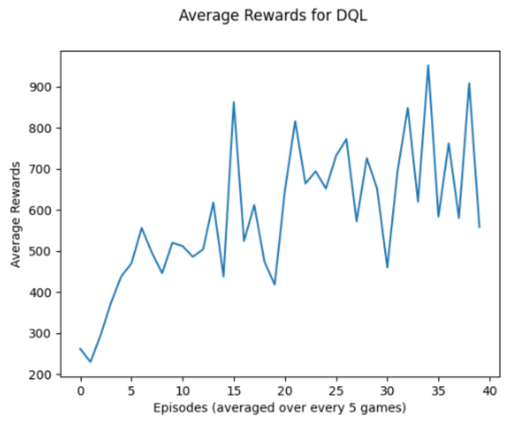

Figure 5: Rewards over 200 games of Ms. Pac-Man (averaged over every 5 games)

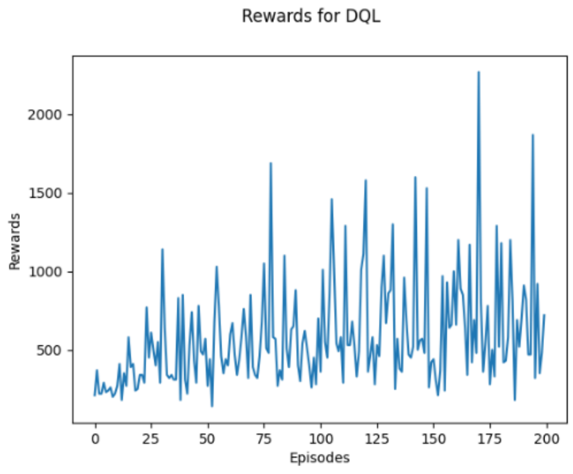

Figure 6: Rewards over all 200 games of Ms. Pac-Man (no average)

Our experiments indicated that having a learning rate of $\alpha = 0.0005$ for the Adam Optimizer and the discount factor of $\gamma = 0.99$ resulted in the best performing model. This is probably because having a small learning rate allows it to make smaller, more precise updates to the weights. Additionally, having a high discount factor forces the Pac-Man to value future rewards more than immediate rewards. We also designed our model to have an epsilon of $\epsilon = 1$ which decays by 0.99995 in each time step until it reaches $\epsilon = 0.05$. This is so that the Pac-Man can explore the environment initially to find the best moves and then take the greedy actions later when it is more familiar with the environment. Finally, we decided to use the Adam optimizer as our optimizer and the Root Mean Square as our loss function due to its popularity and performance in deep neural networks.

When looking at the graphs (Figure 5 and 6), we can immediately see that the rewards fluctuate a lot compared to the SARSA and Q-Learning algorithm (Figure 2 and Figure 3), even though they both hover around 700 rewards per game. Additionally, when compared to the actor-critic algorithm (Figure 4), although they both fluctuate a lot, we can see that the Deep Q-Learning algorithm performs better as it plays more games.

Finally, an observation that we made when testing the Deep Q-Learning algorithm with the best model that achieved the highest reward, is that we've noticed that the Pac-Man will initially take a couple of actions before hitting a wall and then stay there until a ghost has captured the Pac-Man. This is quite similar to what happened with the SARSA and Q-Learning algorithm. Our guess is that the SARSA, Q-Learning, and Deep Q-Learning algorithms need more time to learn the optimal strategies.

3.5 Comparison of Algorithms

Out of all of the RL algorithms investigated in this paper, SARSA and Q-Learning proved to be the most consistent (refer to Figures 2, 3). Both displayed a rapid period of growth in terms of rewards for the first $\tilde{2}0$ episodes, and importantly, remained at a stable reward of $\tilde{7}00$. Deep Q-Learning by contrast, was able to yield much higher rewards during certain episodes (as seen in Figures 5 and 6), but featured a slower growth rate, and overall greater variability. These findings indicate that the traditional methods of Q-Learning and SARSA struggled with higher-dimensional state spaces, whilst Deep Q-Learning did not, at the expense of stability. Finally, Actor-Critic was unable to yield suitable results for the Ms. Pac-Man game. A possible explanation would be that the stochasticity of the game is difficult to take into account.

4 Conclusion

This project has allowed us to compare four different algorithms, SARSA, Q-Learning, Deep Q-Learning, and Actor Critic, to evaluate their performance for solving Ms. Pac-Man (achieving a high score). Although complex Ms. Pac-Man environment has proven to be a challenge for the algorithms discussed, it has showcased the strengths and weaknesses of each one. As Ms. Pac-Man does not have a "goal" state, as well as the stochasticity of the ghosts, we cannot conclude there is "one" approach to the Ms. Pac-Man game. Rather, there exists a multitude of viable approaches, such as the ones discussed in this paper.

Due to the instability given by the Deep Q-Learning algorithm, if we would have more time and/or resources, we would explore other algorithms like proximal policy evaluation (PPO) or Advantage Actor-Critic, which promise better scalability and stability. Finally, in the future we would like to apply Double Q-Learning and Dueling Q-Learning for our Deep Q-Learning algorithm. Additionally, with greater resources, further variations of algorithms could be tested, such as different hyper-parameters.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[3] Lori L. DeLooze and Wesley R. Viner. Fuzzy q-learning in a nondeterministic environment: developing an intelligent ms. pac-man agent. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 162–169, 2009.

[4] Yuanfang Li and Xin Li. Atari breakout with neural networks.

[5] Adam Paszke and Mark Towers. Reinforcement learning (dqn) tutorial, 2024.

[6] Johnny Code. Implement deep q-learning (dqn) in pytorch - beginner reinforcement learning tutorial.