Fall 2021

Playing Pong Using Q-Learning

Akash Kumar

Playing Pong Using Q-Learning

A Thesis

Presented to the Faculty of the

Department of Computer Science

West Chester University

West Chester, Pennsylvania

In Partial Fulfillment of the Requirements for

the Degree of

Master of Science

By

Akash Kumar

December 2021

Acknowledgements

I would like to thank my family for constantly supporting me from the sidelines throughout the course of this project. I would also like to thank my advisor, Dr. Richard Burns, for believing in me and for providing me with his invaluable advice.

Abstract

This thesis involves the use of a reinforcement learning algorithm (RL) called Q-learning to train a Q-agent to play a game of Pong against a near-perfect opponent. Compared to previously related work which trained Pong RL agents by combining Q-learning with deep learning in an algorithm known as Deep Q-Networks, the work presented in this paper takes advantage of known environment constraints of the custom-made Pong environment to train the agent using one-step Q-learning alone. In addition, the thesis explores ways of making the Q-learning more efficient by converting Markov Decision Processes (MDPs) to Partially Observable Markov Decision Processes (POMDPs), and by using state reduction techniques such as state discretization and state distillation. Based on experiments conducted, this thesis highlights that it is possible to use one-step Q-learning, a model-free algorithm typically relegated to solving simple maze world environments, in combination with a POMDP and state distillation to train a Q-agent to play Pong and converge to the optimal policy.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1. Pong and Its History

Pong is a two-dimensional table tennis video game that is about half a century old. It was manufactured by Atari corporations and released in 1972 (Modany 2012). Initially assigned by Nolan Bushnell, founder of Atari, to programmer Al Alcorn as a training exercise, Pong was a huge success and became the first commercially successful video game. In 1975, Atari released a home edition of Pong under the Atari 2600 - the Cartridge Tele-Games System Video Arcade - which sold around 150,000 units. “Today, the Pong Game is considered to be the game which started the video games industry, as it proved that the video games market can produce significant revenues” ("Pong Game" 2021).

Similar to table tennis, the game features two paddles and a ball. The player controls the paddle by moving it vertically up and down across the screen. The screen is vertically divided into two sides, left and right, with each side being occupied by a player’s paddle. Players take turns hitting the ball back and forth using their paddles between their respective sides. The ball’s return angle and speed can be controlled by hitting it at different parts of the paddle. If a player manages to get the ball past their opponent, they score a point and vice versa. The goal of the game is to score 11 points (in some variants 10) before the opponent does and win the game. The game can be played with two human players, or one player against a computer-controlled paddle.

1.2. Overview

The goal of this thesis is to train an agent to play Pong using Q-learning. The setup involves having two Pong Artificial Intelligences (AIs) play against each other, with Player 1 being the Q-learning agent and Player 2 being the near-perfect opponent. Q-learning is a form of reinforcement learning, an area of machine learning that deals with training AI to "become proficient in an unknown environment, given only its percepts [of the environment] and occasional rewards" (Russell 2021). Reinforcement learning (RL) differs from supervised learning in that in supervised learning, the agent learns by passively observing a dataset with pre-labeled inputs/outputs, whereas in RL, the agent actively takes actions in its environment and learns from experience (Russell 2021). To complicate matters, the actions the agent takes alters the state of the agent and/or its environment, making RL harder than supervised learning.

A RL problem consists of three main concepts: the agent, the environment, and the state of the agent in the environment (see Figure 1) (Bhatt 2018). The agent is the intelligent decision maker of the RL problem. It interacts with its environment by making actions and receiving rewards from its environment based on those actions. The environment is the demonstration of the problem to be interacted with and solved by the agent. It can be either a real-world environment or a simulated one. Finally, the state is the current condition of the agent and its environment at a specific time-step. Every time the agent performs an action, the time-step increments and the agent transitions to a new state in the environment (Blackburn 2019). After doing enough actions in its environment, the RL agent will have completed an episode, where one episode consists

of the states that come in between an initial state and a terminal state. Essentially, the goal of the agent is to maximize the rewards it receives in any given episode and it does this by learning the best strategy (optimal policy) to follow by completing multiple episodes in its environment in a process known as training (Zychlinski 2019).

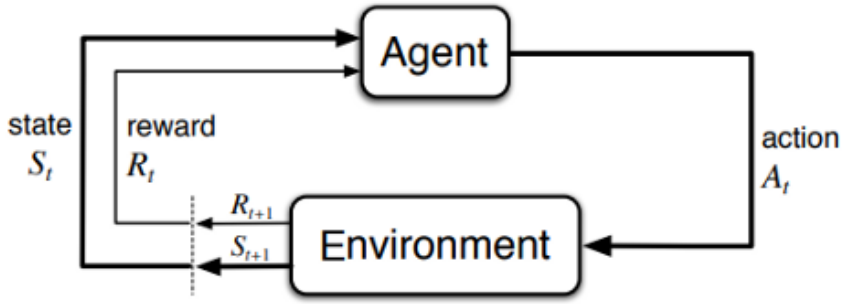


Figure 1: Typical reinforcement learning cycle

When solving a RL problem, there are two main methods to choose from: value iteration and policy iteration. Value iteration involves "calculating the value functions or Q-values of each state and choosing actions according to those" (Zychlinski 2019). Policy iteration involves directly computing a policy "which defines the probabilities each action should be taken depending on the current state, and act according to it" (Zychlinski 2019). While each of these methods can be used as standalone solutions to the RL problem, both value iteration and policy iteration can be combined to create a more robust method called actor-critic algorithms, where the policy function plays the role of the actor and the value function plays the role of the critic (Zychlinski 2019). However, in this thesis, the focus will be on value iteration alone, particularly a specific

kind of value iteration called Q-learning, to which a brief introduction is provided in Section 2.2.

1.3. Outline

This thesis proceeds as follows:

- In Section 2, background theory on Markov Decision Processes (MDPs), Partially Observable Markov Decision Processes (POMDPs), and Q-learning is presented. The components of an MDP as well as the types of Q-learning relevant to this work will be explained.
- Section 3 presents the Q-learning algorithm design and implementations for the Pong AI game that is implemented in this thesis, preceded by a simple maze world Q-learning implementation to test Q-learning theory. Decisions include choosing a programming language, making use of a POMDP Q-table, designing a Q-function implementation, experimenting with reward systems and exploration functions, figuring out how to apply state distillation to the game environment, etc.
- In Section 4, the results of the Q-learning experiments are highlighted and compared. The results of the experiments that yielded optimal policies are first presented, followed by experiments comparing the values of the Q-learning hyperparameters and different exploration functions. Results are compared using measurements such as number of trials, training time,

percent of states visited, and whether or not the Q-agent converged to the optimal policy.

- In Section 5, other work related to this thesis is noted. Topics include learning to utilize shaping rewards in reinforcement learning, Q-learning convergence for non-MDPs, and playing Atari games with deep Q-learning.
- In Section 6, ideas, thoughts, and intentions for improving or expanding on this thesis's work is discussed.
- Section 7 briefly summarizes the ideas and results presented in this thesis.

2. Background

Markov Decision Process (MDP) is fundamental to the idea of Q-learning and is first introduced at the top of Section 2.1. This is followed by a Q-learning primer in Section 2.2 and an introduction to the various types of Q-learning in Section 2.3.

2.1. MDPs and POMDPs

A Markov Decision Process or MDP is "a sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards" (Russell 2021). MDPs provide a way to formulate RL problems mathematically (Blackburn 2019). The components of an MDP are a set of states S beginning with an initial state s_0, a set of actions `Actions`(s) in each state, a transition model P(s' | s, a) which gives the probability of reaching the next state s' given the RL agent takes action a in state s, a reward function R(s, a) which rewards the agent with a certain amount for taking action a in state s, and a policy function $\pi(s)$ which is used to determine the action that the agent will take in any given state in the environment. Methods for solving MDPs usually involve dynamic programming, a type of problem-solving approach where a complex problem is broken down into simpler problems by recursively breaking the problem into smaller pieces and storing the optimal solutions to those pieces for later use (Russell 2021). Before solving any decision problem using the MDP method, one must ensure that the state transitions obey the Markov property. The Markov property states

that the "future is independent of the past given the present" (Blackburn 2019) and can be mathematically expressed as follows:

$$P(S_{t+1} \mid S_t) = P(S_{t+1} | S_1, \dots, S_t)$$

Equation 1: The Markov property

State transitions which obey the Markov property (Equation 1) are said to be Markovian in nature. Having this Markov guarantee of MDPs makes solving decision problems much easier since we do not have to take into account previous states in the state history to predict the future state. The future state S_{t+1} depends only on the present state S_t of the environment and is entirely independent of the past (Blackburn 2019).

A subset of the MDP called the Partially Observable Markov Decision Process or POMDP extends on the concepts of the MDP. Unlike a regular MDP which involves making decisions in a fully observable environment, a POMDP involves making decisions in a partially observable environment (Hansen 2013). A POMDP is used for creating an elegant mathematical model for "planning and control problems for which there can be uncertainty about the effects of actions and about the current state" (Hansen 2013). In a partially observable environment, certainty or determinism of the environment is lost, so small changes in the past can make big differences to the state in the present and future. Since the future state now depends on the condition of the present as well as past states, this violates the Markov property. However, there is a solution to this complication. "It is well-known that a state probability distribution updated by Bayesian reasoning is a sufficient statistic that summarizes all information about the

history of the process necessary for optimal action selection" (Hansen 2013). In other words, by combining a state probability distribution with Bayesian reasoning, we preserve the Markov property of the POMDP. Given that the Markov property holds true, the POMDP can be recast into a standard completely observable MDP with a state space that consists of all possible state probability distributions (Hansen 2013). Note here that sometimes, a POMDP can already behave like an MDP. This typically happens in cases where only those states which do not impact the state transitions of the agent are ignored. Furthermore, if such state transitions are deterministic in nature (instead of stochastic), then the transition model of the MDP also becomes deterministic, and a demonstration of such a case is exemplified in Section 3.5 with the POMDP Q-table.

2.2. Q-Learning Primer

Q-learning is a model-free, off-policy reinforcement learning algorithm (Russell 2021). Model-free means the agent does not learn a transition model for the environment and off-policy means the agent learns the value of the optimal policy independently of its actions. The 'Q' in Q-learning stands for quality (Shyalika 2019). As an asynchronous dynamic programming method, "it provides agents with the capability of learning to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains" (Watkins 1992). Like any RL algorithm, learning occurs through trial and error. The Q-learning agent or Q-agent learns by taking actions at states and evaluating the Q-values (quality values) of such state-action pairs

based on the reward or penalty it receives. Q-values are used to calculate the maximum expected future rewards for every action at each state. These Q-values are stored in a lookup table data structure called a Q-table to be used in the future by the Q-agent for deciding the best action to take in a given state (Shyalika 2019). The framework is shown in Figure 2. The use of the Q-table is similar to the idea of memoization in dynamic programming, where for complicated systems with a large number of states a considerable set of possible decisions are stored for future use (Bellman 1966, Torres 2020). "By trying all actions in all states repeatedly, the Q-agent learns which are best overall, judged by long-term discounted reward" (Watkins 1992). It has been proven that as long as all actions are repeatedly sampled in all states of the environment and all action-values are discrete, the Q-learning will converge to the optimal action-values with probability 1 (Watkins 1992). In this sense, Q-learning is a Monte-Carlo method, where even if the state transition probability P is not known, after many tries, the final estimated Q-value of any given state-action pair will be very close to its real Q-value (Lee 2019a).

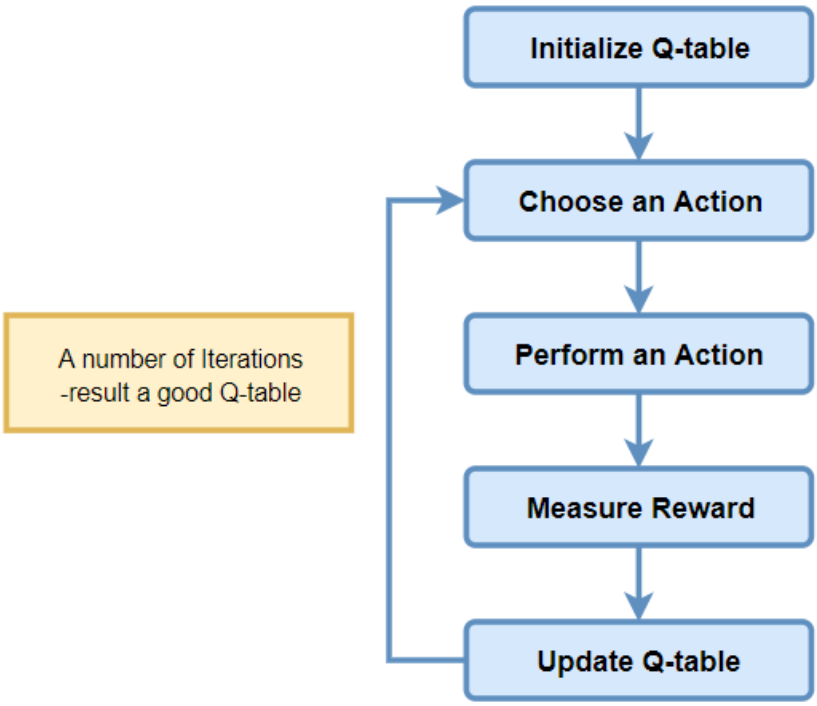


Figure 2: The Q-learning process

The Q-values for any given state-action pair is calculated using the Q-function. The Q-function is based on the Bellman equation, an equation named after Richard Bellman which calculates the value or utility of a state in terms of the immediate reward from that state plus the maximum discounted future utility by taking the best action in that state.

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma U(s')]$$

Equation 2: The Bellman equation

In Equation 2, the utility of the current state *s* is calculated by combining the reward of taking the best action *a* in state *s* with the discounted utility of the future state γU(*s*'). Modifying the Bellman equation slightly, we get the Q-function equation:

$$Q(s,a) = \sum_{s\prime} P(s'|s,a)[R(s,a,s') + \gamma\, U(s')]$$

$$= \sum_{s\prime} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q(s',a')]$$

Equation 3: The Q-function equation

Similar to Equation 2, the Q-function equation (Equation 3) calculates the Q-value of the current state *s* by combining the reward of taking action *a* in state (where *a* is not necessarily the optimal action) with the discounted Q-value of taking the optimal action *a*' in the next state *s*'.

2.3. Types of Q-learning

The four main types of Q-learning are: one-step Q-learning, TD Q-learning, deep Q-learning, and double Q-learning.

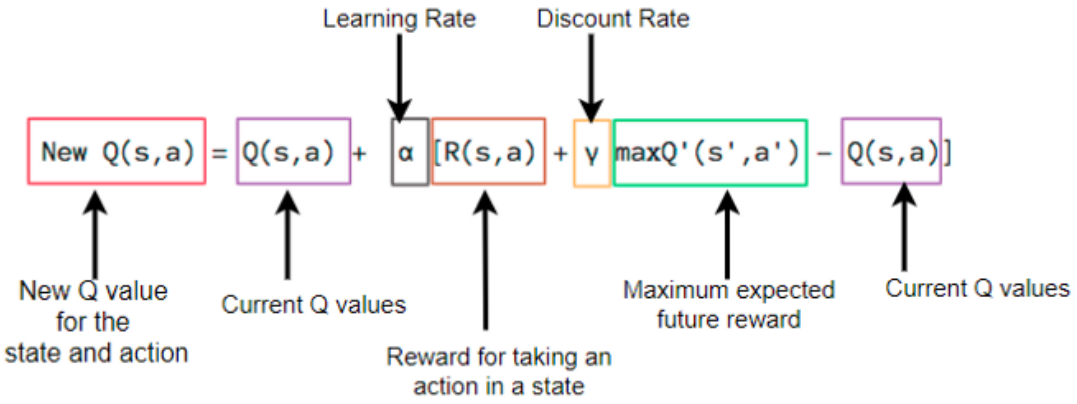
One-step Q-learning is the simplest kind of Q-learning (Peng 1994, Watkins 1992). At each state in its environment at timestep *t*, the Q-agent is only concerned with the utilities of actions in that state which will get it to the next state *s*' in the next timestep *t* + 1, hence the name one-step Q-learning. In addition, unlike the Q-function equation

presented in Equation 3, the one-step Q-function ignores the transition model P(*s*' | *s*, *a*). This is because the environment is assumed to be deterministic instead of stochastic, so a transition model is not required. The equation for the one-step Q-function is presented in Equation 4. $\delta(s, a)$ in Equation 4 represents the future state *s*' after having taken action *a* in state *s*.

$$\hat{Q}(s,a) = r(s,a) + \gamma \max_{a'} \hat{Q}(\delta(s,a), a')$$

Equation 4: One-step Q-function equation

Next, we have the Temporal-difference Q-learning or TD Q-learning for short. Originally called incremental multi-step Q-learning (Q(λ)-learning) by its inventor Dr. Jing Peng, the TD Q-learning algorithm "extends the one-step Q-learning algorithm by combining it with TD(λ) returns for general λ in a natural way for delayed reinforcement learning" (Peng 1994). TD Q-learning has a number of advantages over one-step Q-learning. For one thing, since corrections are made incrementally to the predictions of observations occurring in the past, TD Q-learning propagates information faster to where it is important. Second, TD Q-learning works significantly better than one-step Q-learning on a number of tasks (Peng 1994). Third, TD Q-learning, unlike one-step, has been shown to converge well on stochastic environments and even some kinds of non-MDP problems (Majeed 2018).



Equation 5: TD Q-function equation

The TD Q-function equation presented in Equation 5 highlights the different parts of the equation. Similar to the one-step Q-function equation, the discount factor γ (discount rate, typically strictly less than 1) is used to discount the Q-value of the future state before adding it to the reward. By giving a smaller weight to Q-values further down the state sequence, rewards obtained closer to the present state are seen as more valuable than those obtained in the future (Ng 2003). For low discount factors close to 0, the agent prefers immediate rewards over future rewards and for high discount factors close to 1, the agent prefers future rewards over immediate rewards. In addition, the TD Q-function equation also includes a learning rate and a TD error. The learning rate α is used to control the rate of the Q-learning. A low α will result in slow learning and a high probability of converging to the optimal policy, while a high α will result in fast learning and a low probability of converging to the optimal policy (Russell 2021). The TD error is the temporal difference between the maximum expected future reward and the current Q-

value (Lee 2019b). This temporal difference between utilities of successive states is what gives TD learning its name (Russell 2021). Another key distinction between the one-step and TD Q-functions is that while the one-step Q-function overwrites previously calculated Q-values for state-action pairs, the TD Q-function uses the values of the previous Q-values to calculate the new Q-values, resulting in less loss of information.

A more advanced kind of Q-learning is deep Q-learning. Deep Q-learning uses a Deep Q-Network (DQN) to train the Q-agent. DQN is a convolutional neural network (CNN) trained with a variant of Q-learning that is used as a non-linear function approximator for the Q-function (Mnih 2013). Rather than using a lookup Q-table to store Q-values for state-action pairs like in one-step Q-learning, Q-values are calculated using combinations of weights and biases in the network. That is, instead of explicitly storing Q-values for state-action pairs, the DQN implicitly calculates them. This method of calculating Q-values has a few benefits over the previous methods: 1) less memory is required for large state spaces, 2) the Q-learning can be used on continuous state spaces rather than just discrete ones, 3) the state can be represented using images instead of explicit state attributes thanks to the use of CNNs, and 4) learning is faster for large state spaces as the Q-agent generalizes its learning and does not need to try every possible state-action pair in its environment multiple times in order to learn the optimal policy (Mnih 2013). However, one of the major downsides of DQNs is that the variance in the network updates may be high due to strong correlations in consecutive samples of the RL data. High variance in turn leads to poor and inefficient learning. To counteract this issue, a technique known as experience replay, where the agent's previous experiences are

stored in a replay buffer and replayed randomly to the agent, is used (Mnih 2013). Further details regarding experience replay and the DQN algorithm can be found in Section 5.3.

Another type of Q-learning that is fairly unknown is double Q-learning. Double Q-learning is based off of TD Q-learning and, like the name implies, uses two Q-functions instead of one. While TD Q-learning uses a single TD Q-function to estimate the Q-value of the next state, double Q-learning uses two TD Q-functions, Q^A and Q^B, to estimate the Q-values (see Equation 6). Each of the two functions updates the current state-pair's Q-value with a value from the other Q-function for the next state (Hasselt 2010). That is, Q^A calls Q^B and Q^B calls Q^A in a kind of mutual recursion. The advantage of double Q-learning over TD Q-learning is that it performs better in some stochastic environments. This is because TD Q-learning (and one-step Q-learning) makes large overestimations of action values. "These overestimations result from a positive bias that is introduced because Q-learning uses the maximum action value as an approximation for the maximum expected action value" (Hasselt 2010). In contrast, the double estimator method has been shown to sometimes underestimate rather than overestimate the maximum expected value. This underestimation of expected values means that double Q-learning will often converge faster than regular Q-learning in certain stochastic environments. Moreover, the overestimation tendencies which plague regular Q-learning also affect DQNs. It has been shown that double Q-learning can be combined with a DQN to create a double DQN which not only results in more stable and reliable learning but also helps the Q-agent find better policies (Hasselt 2016).

$$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)(r + \gamma Q^B(s', a^*) - Q^A(s,a))$$

$$Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s', a^*) - Q^B(s,a))$$

Equation 6: Double Q-learning function equations

In this subsection, four main types of Q-learning were discussed: one-step Q-learning, TD Q-learning, deep Q-learning, and double Q-learning. Of these types, one-step Q-learning is the one that is primarily used in this thesis as it is sufficient to tackle the deterministic nature of the Pong RL environment. Next in Section 3, we will show the design and implementation of our Q-learning algorithm.

3. Q-Learning Design

In this section, we present the Q-learning algorithm designed to train a Pong Q-agent to play against a near-perfect opponent. We will describe the thought process, design decisions, and challenges encountered while creating the Q-learning algorithm.

3.1. Simple Q-Learning Implementation

This subsection presents a proof-of-concept of the Q-learning fundamentals that will be extended later in Sections 3.4 and 3.9. This simple implementation, taken from Russell 2021, consists of a 4 x 3 maze environment which is represented using a 2-dimensional list in Python and is shown in Figure 3. The maze has two terminal states each with a reward of +1 and -1 at coordinates (4, 2) and (4, 3) respectively. Note that the coordinates are written as columns first and rows second. In addition, there is a negative reward $R(s,a,s') = -0.04$ for transition between states. This is to encourage the agent to exit the environment using the fewest moves possible. The goal is to use Q-learning to make the Q-agent learn the optimal path from the starting state to one of the terminal states, with the optimal path being not necessarily the shortest path, but the path which maximizes positive rewards and minimizes negative rewards. The policy used by the agent to follow the optimal path is known as the optimal policy and is shown in Figure 3a (Russell 2021).

Figure 3a: Suboptimal paths for the 4 x 3 maze world

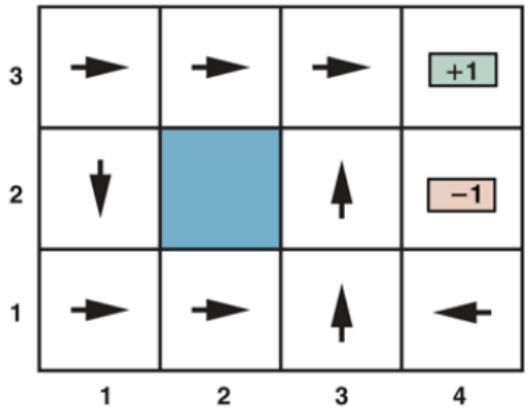
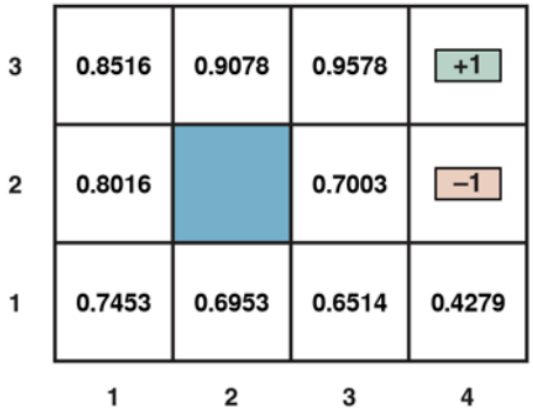


Figure 3b: State utilities of the 4 x 3 maze world



Given enough trials of the maze world, the agent will eventually learn the optimal policy of its environment. The optimal policy is defined based on the utility of the environment. Each state in the environment has a utility value attached to it. Each trial conducted provides a sample of the utilities of the states visited. “In the limit of infinitely many trials, the sample average will converge to the true expectation”. This is the idea behind direct utility estimation (Russell 2021). In Figure 3a, the Q-learning converged to an optimal path from state (1,1) to the terminal state (4,3), with the optimal path being (1,1) → (2,1) → (3,1) → (3,2) → (3,3) → (4,3). In fact, if it starts at any of the other non-terminal states, it will still reach the positive terminal state. All of these paths are optimal except for the path starting at state (1, 2), which is suboptimal (the optimal direction is up, not down).

Figure 3b shows the learned utilities of all the states in the 4 x 3 maze world learned via Q-learning. Originally, the idea was to store the Q-values for the world in a Q-table of dimensions (4 x 3) x 4, since there are 12 states in the world and each state has four possible actions (up, down, left, right). This results in a 3-dimensional Q-table containing 48 different Q-values (12 x 4 = 48). While this table worked in making the Q-agent learn the optimal path, we soon realized a 4 x 3 Q-table was enough to do the job. This is done by implicitly storing the state-action-pairs of the Q-function. In other words, while the states are explicitly defined in the Q-table, the actions taken to move between the states are not. This is possible to do because in the implementation, the states and actions are represented as indices (or coordinates) of the maze/Q-table. So, by adding the action's coordinates to the state's coordinates and only allowing valid moves, we can move the agent through the maze. In this way, the state-action pairs can be reduced to future states and the utility values for those states can be stored in the Q-table. The TD Q-function used to do this is presented in Algorithm 1:

```
# The Q-value of the current state is based on the max Q-value of the next state.
next_state_max_q = max(next state's Q-values)
qtable[s[0]+a[0]][s[1]+a[1]] = (qtable[s[0]+a[0]][s[1]+a[1]] + α * (R(s,a) + γ * next_state_max_q -
        qtable[s[0]+a[0]][s[1]+a[1]]))
return qtable[s[0]+a[0]][s[1]+a[1]]
```

Algorithm 1: Maze world Q-function for reduced Q-table

Using the reduced 4 x 3 Q-table (or state Q-table), after 25 trials, the Q-agent converges on the optimal policy. Instead of a Q-table containing 48 Q-values (4x3x4), we have one that has only 12 Q-values (4x3), the same number as the total number of states in the maze world (including invalid states such (2,2) which is not reachable by the agent). By reducing the size of the Q-table by a factor of 4, we store fewer Q-values. Since the agent now needs to learn a fewer number of Q-values, training is faster and less memory is used. Experiments were conducted in Python comparing the training time and memory space usage of the regular and reduced Q-tables and are shown in Table 1. Based on these measurements, compared to the regular Q-table, the reduced Q-table resulted in 4 times less trials required, 1.79 times faster training time, and 4.55 times less memory usage. Thus, we have taken a 4 x 3 x 4 Q-table and distilled it down to a 4 x 3 Q-table which accomplishes the same thing but with better performance. For environment sizes much larger than 4 x 3, one can expect the performance differences to be more pronounced. Further in the paper, we will come across an idea called state distillation which works in a similar way to the reduced Q-table.

Q-table type	**Number of trials**	**Time taken to train (in seconds)**	**Memory usage of Q-table (in bytes)**
Regular Q-table (state-action Q-table)	100	0.0179994	3384
Reduced Q-table (state Q-table)	25	0.0100358	744

Table 1: Regular Q-table vs. Reduced Q-table

3.2. Pong Environment Attributes and Constraints

The pong environment in this thesis has the following attributes:

- Attributes of the Pong canvas:
    - width: 1,400 pixels (px)
    - height: 1,000 px
- Attributes of the ball:
    - width: 18 px
    - height: 18 px
    - starting x-value: 700 px
    - starting y-value: 500 px
    - x-speed: 9 px / frame
    - y-speed: 6 px / frame
- Attributes of the paddle:
    - width: 18 px
    - height: 70 px
    - fixed x-value: 150 px for left paddle and 850 px for right paddle
    - starting y-value: 465 px
    - speed: 6 px / frame
- Frame rate: 18 – 40 frames per second

The constraints of the Pong environment are as follows:

- The winning score of the Pong game is 11.
- The paddles only move in the y-direction. Their x-position is fixed.
- The paddles do not go outside the boundaries of the Pong canvas.
- The ball does not go outside the top and bottom boundaries of the Pong canvas, but is allowed to go outside the left and right boundaries.
- When the ball hits the top of the game canvas, its y-direction reverses from up to down. When it hits the bottom of the canvas, its y-direction reverses from down to up. When it hits the left paddle, its x-direction reverses from left to right. When it hits the right paddle, its x-direction reverses from right to left.
- Where the ball hits the paddle does not affect the speed or angle of the ball. The ball always bounces off the paddles at a fixed angle at a fixed speed.
- The speed of the paddles in the y-axis is the same as the speed of the ball in the y-axis.
- The left paddle is controlled by the Q-agent. The right paddle is pre-programmed to follow the ball in the y-axis and center the y-position of its center to the y-position of the ball's center.
- When the ball goes past the left or right boundaries of the canvas, the Pong rally ends and a point is awarded to the winner. In the next rally, the

ball is initialized on the loser's side and moves in the direction of the loser's paddle.

- At the start of each rally, both paddles start at the center of the canvas's height.

3.3. Pong Environment and Implementation Details

Originally, we created a Pong game environment in JavaScript. The environment consisted of a game canvas, a ball object, two paddle objects, and a scoreboard. These objects consisted of attributes such as `x-position`, `y-position`, `width`, `height`, `moveX` (ball), `moveY` (ball and paddle), `speed`, and `score`. These attributes could be changed as needed to make the game easier or harder for the AI. All of the objects of the Pong game were constantly updated in a game loop until either of the two players got the winning score (default winning score is 11). The ball moved at a fixed speed. Each paddle had 3 possible actions: idle, up, and down and can only move in the y-axis. For testing, we programmed both paddles to align their centers with the y-coordinate of the ball so as to get the paddles to follow the ball in the y-axis. Note here that the speed of the ball and the paddles in the y-axis is the same.

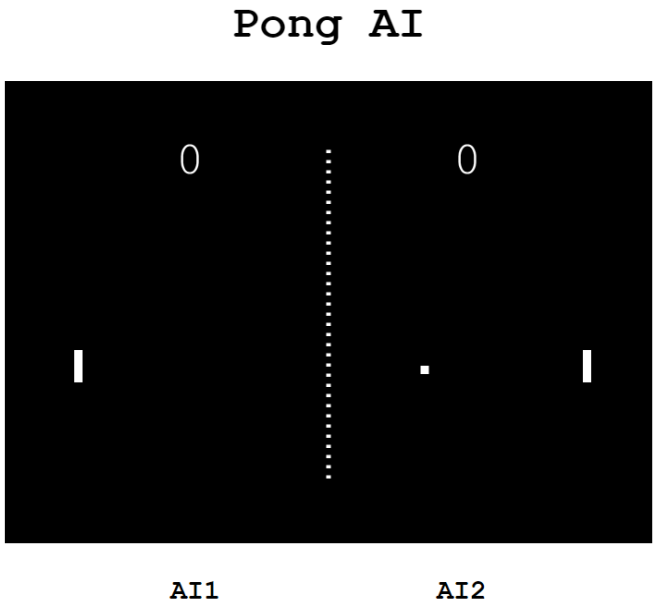


Figure 4: Pong AI game in web browser

While the Pong game rendered fine in the browser, it was found to be quite limiting for experimentation. For one thing, the game could only be rendered on a browser. Second, JavaScript is an “asynchronous, blocking, single-threaded language”, meaning it can only run one thread at a time ("Introducing Asynchronous Javascript - Learn Web Development | MDN" 2021). This makes it hard to do parallel programming in JavaScript and whatever processes are parallelizable have to be IO-bound. Due to these reasons, we opted to create a new Pong AI environment in Python instead.

So, we ported the JavaScript code over to Python3. Objects which were previously created using functions in JavaScript were instantiated using classes in Python. Those classes are: `Ball`, `Paddle`, `PongGame`, and `Qagent` (see Table 2). The attributes for these classes are listed in Section 3.2. As before, the entire Pong game and all its objects are still being updated through the main loop and all the attributes and

dimensions of the objects can be changed as needed. However, there was one caveat with the Python Pong AI implementation: there was no GUI to display or view the game with.

Class	**Purpose**
`Ball`	to create a ball object.
`Paddle`	to create a paddle object.
`PongGame`	to create a Pong canvas and simulate the Pong game.
`Qagent`	to create a Q-agent which plays the Pong game.

Table 2: Classes of Python Pong Environment

While performing computations and changing the program's variables alone was enough to play the Pong game, there was no visual way to view the actual game and see whether the paddles and ball were behaving as expected. This made debugging the game similar to a black box. On the bright side, the fact that the game could be played purely using computations markedly sped up the training time. With the Pong GUI being displayed, the average time it took to finish one match of Pong (11 games/trials) was 27.340s. Without displaying the GUI, one match of Pong took 1.314s to finish (see Table 3). That is a remarkable 95.19% faster training time!

	Average Time Taken to Complete One Match (in seconds)
Pong Game with GUI Included	27.340s
Pong Game with GUI Excluded	1.314s

Table 3: Effect of Python Pong GUI on training time

To solve this above issue, a Pong GUI was created and added for the Python implementation using a Python library called Pyglet. Pyglet is a cross-platform windowing and multimedia library for Python. It is primarily meant for developing games and graphical window applications on Microsoft Windows, Mac OS X, and Linux. It is free to use, open-source, and was released under the BSD license ("Pyglet" 2021). An alternative GUI library called Pygame was also considered, but after considering that Pygame is slower than Pyglet and is less flexible, Pyglet was used instead (Shinners 2000).

When creating the Pyglet Pong GUI (shown in Figure 5, where the left paddle is the Q-agent and the right paddle is the near-perfect opponent), much of the previous Python code was left largely unchanged. One of the main changes that was made was changing the `PongGame` class so that it inherits from `pyglet.window.Window`. In fact, all of the Pyglet code and its corresponding methods were contained inside the `PongGame` class. This is so that the objects of the other classes do not have control over nor do they have to concern themselves with the workings of the Pyglet app. Another

change made was to include the current action taken by the Q-agent (AI1) in the Pyglet GUI display. Each of the three actions of the paddle is associated with an index: IDLE (0), UP (1), and DOWN (2). This indexing of the actions is useful for storing and accessing the values of actions in the Q-table. The index is displayed along with the action for clarity, particularly in cases where the Q-agent rapidly switches between actions because of uncertainty.

Figure 5: Pyglet Pong GUI

3.4. Q-Table Implementation

Having set up the Pong environment and the Pong GUI, the focus now shifts to the implementation details of the Q-agent, particularly in this section, the agent's Q-table. The RL Pong implementation in this thesis considers the following seven state variables:

1) the y-position of the Q-agent's paddle.

2) the y-position of the opponent's paddle.

3) the x-position of the ball.

4) the y-position of the ball.

5) the x-direction of the ball (left, right).

6) the y-direction of the ball (up, down).

7) the action of the Q-agent's paddle (idle, up, down).

Of these seven variables, only three are necessary for the Q-agent to learn and perform well in its environment, namely 1), 4), and 7). This is because the paddle moves at the same speed in the y-axis as the ball does. Also, where the ball hits the paddle upon contact does not affect the angle or speed of the ball. In other words, the ball's speed both in the x and y-directions remains constant. When the paddle hits the ball, only the x-direction of the ball reverses (left becomes right, right becomes left). Likewise, when the ball contacts the top and bottom walls of the Pong canvas, the y-direction of the ball reverses (up becomes down, down becomes up). Thus, due to these known environment constraints highlighted previously in Section 3.2, it should be possible to use a partially observed state rather than having to use the full state to train the Q-agent. Furthermore, since the unobserved states of the environment do not impact the Q-values of the observed states, the environment is still deterministic and the POMDP can be treated as a regular MDP.

Given the dimensions of the Pong canvas as well as the possible positions of the objects contained in it, with a fully observed state, the Q-table will have the following seven dimensions:

$$1000_{\text{paddle y-positions}} \times 1000_{\text{opponent paddle y-positions}} \times 1400_{\text{ball x-positions}} \times 1000_{\text{ball y-positions}} \times 2_{\text{ball x-directions}} \times 2_{\text{ball y-directions}} \times 3_{\text{paddle actions}}$$

or 16.8 trillion Q-values! This is simply infeasible to train within a reasonable span of time.

With the partially observed state, the three dimensions of the Q-table will be

$$1000_{\text{paddle y-positions}} \times 1000_{\text{ball y-positions}} \times 3_{\text{paddle actions}}$$

or 3 million Q-values. While the complexity of the MDP is polynomial in the size of its parameters, the size of the state space is exponential in the number of dimensions of the state vector (Arora 2021). It is clear from these dimension estimates that by training the Q-agent on the partially observable state, we use a POMDP Q-table that is an order of 5.6 million times smaller than the Q-table based on the fully observable state. Not only is the reduced Q-table more memory efficient, but the Q-agent has much fewer Q-values to learn, resulting in exponential savings in training time.

3.5. Q-Function Implementation

The Q-function $Q(s, a)$ used in the Pong environment is based on the one-step Q-function introduced in Section 2, though note that the TD Q-learning will also work for

this use case. Given a state *s* and an action *a*, the next state *s*' is calculated. The partially observed state *s* is represented as a tuple of the ball's y-position and the Q-agent paddle's y-position. The action *a* is represented using an action index (0, 1, 2). Using the action's index, we can get the resulting change in y-value of the paddle taking that action will cause. By adding the change in the y-value with the paddle's current y-value, we can get the paddle's future y-value. Similarly, the Pong ball always moves up or down in the y-axis and its y-value is automatically updated based on its movement direction in the y-axis. By adding the change in the y-value of the ball with the ball's current y-value, we can predict the ball's future y-value, and thus know the future y-values for both the ball and paddle.

Combining these y-values of the ball and paddle, we effectively get the future partial state of the Pong environment. This future state in turn can be referenced in the Q-table to return the max Q-value of the next state's Q-values. Finally, the max Q-value is propagated into the Q-equation along with the reward to get the Q-value for the current state-action pair. The Q-function described here is presented in Algorithm 2:

The Q function Q(*s*,*a*) gives the quality of taking action *a* in state *s*. s' is the future state.

s' ← combine action a with state *s*.

next_state_max_q ← max(qtable[s'])

One step Q-value equation for deterministic environment:

qtable[*s*][*a*] = r(*s*, *a*) + γ * next_state_max_q

return qtable[*s*][*a*]

Algorithm 2: Pong AI Q-function

3.6. Reward Systems

There are a number of possible reward systems we can employ to encourage the Q-agent to play well. A reward function r(*s*, *a*) uses a reward system to assign numerical rewards to certain state-action pairs of the RL environment. The default reward system is used to reward the Q-agent the points it gains and/or loses. For example, if the agent makes the ball go past the opponent's paddle, the agent's score increases by 1 and so, it will receive a reward of +1. But if the ball goes past the agent's paddle, the opponent's score increases by 1 and so, it will receive a reward of -1. This reward system should, in theory, work because Pong is a zero-sum game. However, in practice, this reward system does not work as the rewards are too sparse. The agent only gets rewarded at the end of its gameplay when it scores or loses a point. That is, the action right before the terminal state gets rewarded, but all the other actions that come before it do not receive any reward. While it is true that the rewards from the terminal states will eventually backpropagate to the initial states as the number of training sequences tends towards infinity, it can take an impractical amount of time for this to happen, particularly in the case of a lookup Q-table. Additionally, considering the fact that rewards are discounted

more the further away a given state is from the source reward, the idea of training the Q-agent using this reward system will seem impossible.

Since the default reward system is infeasible for training the agent using one-step Q-learning, we need to consider alternative reward systems. Preferably, we want the new reward system to be filled with frequent rewards to avoid the above-mentioned disadvantages that come with a sparse reward system, and instead utilize pseudorewards. Pseudorewards are "fake" rewards that are supplied not by the game, but by a model representation of the game. "Pseudorewards can potentially be used to guide the agent to the optimal policy", resulting in fewer number of trials required and faster training time (Krueger 2018). For example, a human playing Pong intuitively knows that while hitting the ball with the paddle does not give them an in-game reward, nonetheless it still puts them on track to win the game. That is to say that hitting the ball with the paddle is a subgoal to the end goal of getting the ball past the opponent's paddle. While achieving the end goal gives the agent a reward, achieving the subgoal does not. Unlike the Q-agent, when humans hit the ball with their paddle, they receive a small pseudoreward in the form of a positive emotion (Krueger 2018). In order for the Q-agent to receive pseudorewards, we must provide small rewards in the form of numbers when the agent achieves subgoals in addition to the real rewards the game already provides. This way of supplying the agent with pseudorewards as a way of introducing background knowledge into model-free reinforcement learning algorithms is known as reward shaping (Asmuth 2008, Hu 2020). The reason that we can include pseudorewards to the Pong game and potentially modify the original reward function without negative consequences to the

agent's behavior is thanks to the shaping theorem. "According to the shaping theorem, conditions exist under which the optimal decision policy will remain invariant to such modifications of the reward function, opening the possibility that pseudorewards can be used to guide agents toward optimal behavior" (Krueger 2018). As long as the optimal policy remains unchanged after adding the pseudorewards, the agent will converge on that policy.

These are a number of potential pseudoreward systems one could implement to guide the Q-agent to the optimal policy, a policy if followed by the agent will result in the agent winning against the near-perfect opponent 100% of the time. These are described below with the following names: *binary reward system*, *absolute difference reward system*, and *combined reward system*.

Binary reward system: will reward the Q-agent with a reward of +1 whenever the ball is within the y-range of the agent's paddle and a reward of 0 otherwise. This might seem like a good reward system at first, but in practice, the learning is not as efficient as it could be. In most of the states, the ball is not within the y-range of the paddle. In these states, the agent will always receive a reward of 0. Even if the Q-agent chooses the optimal action in these states (e.g., moving the paddle up so as to get closer to the ball in the y-axis), it does not get rewarded for it. So, the agent does not know whether the action it took was a good action or a bad action. It is only after a series of continuous, random, and improbable up/down actions will the ball fall within the y-range of the paddle and the Q-agent will receive a positive pseudoreward of +1. As one can imagine, it can take a long time for the agent to make the necessary random actions to get to the state with the

pseudoreward, and it can take even longer for that reward to backpropagate to the initial state-action pair in the Q-table. While this pseudoreward system is definitely better than the original in-game reward system, the rewards are still not dense enough for efficient learning.

Absolute difference reward system: revolves around the idea of rewarding the agent more the better the paddle's center is aligned with the ball's center. When the paddle is perfectly aligned with the ball, it receives a maximum reward of +1 and the greater the absolute y-difference between the paddle and ball centers, the smaller the reward, eventually approaching zero in the limit of infinite y-difference. In this approach, the Q-agent is rewarded for every possible state-action pair, making this a reward-dense reward system. Since the rewards are plentiful, this greatly helps to make the agent learn the optimal policy much faster and with fewer number of trials compared to the binary reward system. However, how much faster or how many fewer trials exactly is not known as the Q-agent based on the binary reward system did not manage to converge to the optimal policy given the time constraints of the project.

Combined reward system: a variant reward system which combines the ideas in the binary and absolute difference reward systems involves punishing the agent with increasingly negative rewards the further away the paddle is from the ball in the y-axis and giving increasingly positive rewards the closer the ball's center is within range of the paddle's center. This reward system was not tested, but given that most of the state-action pairs in this reward system receive negative rewards and that after enough training the agent will tend to take the action that results in the least negative reward, one can surmise

that this reward system will result in similar performance to the absolute difference reward system.

In addition to the above-mentioned reward systems, a number of other possible reward systems exist for the Pong game. However, not all of these will help the Q-agent converge to the optimal solution. Only those pseudorewards which keep the original optimal policy intact are valid according to the shaping theorem (Krueger 2018).

3.7. Exploration Functions

Exploration functions are used to help the reinforcement learning agent explore its environment. The simplest exploration function is the *random action* function, where at every possible state the agent takes a random action. This method of exploring is the least efficient as the agent does not take past experience and rewards into account when choosing its next action. In other words, the agent is exploring its environment but is not exploiting what it already knows about its environment. While it is theoretically possible for the agent to explore its entire environment using only random actions, for large state spaces, this becomes practically improbable.

In contrast, there exists another exploration function called the *greedy exploration* function. An agent which uses this exploration strategy is called a greedy agent (Russell 2021). Compared to the random action function, greedy exploration takes past rewards and experience in previously visited states into its action-making decisions. However, the downside of the greedy agent is that it is too greedy. Under this exploration strategy, “the

agent always selects the action that [has] the highest state-action value" (Coggan 2004). That is, the agent is only engaging in pure exploitation of the environment and in doing so, fails to fully explore its environment. Without fully exploring the environment, the agent may miss out on exploiting optimal states in the environment and so, will play sub-optimally. This is akin to a person ordering the same meal every time they go to a restaurant without taking the risk to explore the other potentially better options on the menu. "Sometimes greed pays off and the agent converges to the optimal policy, but often it does not" (Russell 2021). So, the takeaway here is that too much exploration is bad and too much exploitation is equally bad for the agent. To effectively explore the environment while taking advantage of previous knowledge of such an environment, the agent must learn to balance between exploration and exploitation. This trade-off between exploration and exploitation is commonly known in the RL literature as the exploration-exploitation trade-off or the exploration-exploitation dilemma (Rocca 2021). A good exploration function should take this dilemma into account so as to maximize the chances of the agent finding the optimal policy of its environment in the shortest time possible.

The *ϵ-greedy* (epsilon-greedy) exploration function is one such potential candidate. "ϵ-greedy algorithm is the most naive (yet efficient!) way to introduce exploration in a decision making process. The idea is simply to take either the action that seems to be optimal with probability (1 - ϵ) (exploitation) or a completely random action with probability ϵ (exploration)" (Rocca 2021). In the initial stages of training, the value of ϵ starts at 1. As the training progresses, the value of ϵ is slowly reduced to 0 or some minimum ϵ-value (like ϵ = 0.1) in a process known as ϵ-decay (epsilon-decay). ϵ-decay is

used so to make the Q-agent focus on exploration in the initial stages of the training when it knows little about its environment and focus on exploitation in the later stages of training to fully learn the optimal policy. While using the ϵ-greedy strategy does help the agent eventually converge to the optimal policy, it can be quite slow (Russell 2021). The agent spends an unnecessary amount of time and effort exploring previously explored bad states (states of low utility) again and again.

An exploration strategy that combines the benefits of the above strategies would involve giving some weight to actions/states the agent has not tried very often while avoiding those actions/states which have proven to be of low utility in the past (Russell 2021). An example of such an exploration strategy would be the *weighted stochastic exploration* function. In this function, actions in states are given weights according to how often such actions have been tried as well as their utility.

The Pong game Q-agent uses this exploration function in combination with the ϵ-greedy function to explore the Pong environment. At each state, weights are attached to the Q-agent's three actions (idle, up, down). The agent is programmed to visit each state in the environment at least *n* times. Actions which lead to states which were visited less than the required minimum visits per state are given a weight of 10; optimal actions (actions which have the max Q-value in their respective states) are given a weight of 1; actions which lead to states of low utility value which have been visited at least n times are given a weight of 0 by default. Once weights have been assigned to all the actions in a given state, an action is chosen at random from the weight distribution of those actions. Since ϵ-greedy is combined with weighted stochastic exploration, the Q-agent takes the

chosen action with probability ϵ (exploration). Otherwise, it exploits the Pong environment based on known Q-values with probability 1 - ϵ (exploitation). The reason ϵ-greedy is combined with weighted stochastic exploration is to reduce the probability of the bad states.

There is one more exploration function that was considered in the design of the Q-learning agent: a *weighted deterministic exploration* function. This function is very similar to the *weighted stochastic exploration* function except that in the initial stages of training it prefers unexplored actions to optimal actions. Once states have been visited at least *n* times, the agent will only visit those states again if they are of good utility. At first appearance, it seems like the deterministic approach might work and even prove to be more efficient than the stochastic approach. Unfortunately, this is not the case. Due to the deterministic nature of the exploration function, the function behaves in a similar respect to the greedy exploration function. Under this exploration strategy, in the initial stages of training, the agent only explores unvisited states. However, due to the low floor for minimum number of visits required per state and the deterministic nature of the exploration, the Q-agent prematurely stops exploring states and focuses only on exploitation of its environment (Vedpathak 2019). This results in the agent exploring only a small percentage of the state space and converging the policy to a bad local optima where the Q-agent always loses against its opponent. Perhaps with a high enough state visit floor, the agent might converge to the optimal policy, but more research and experimentation is required to prove or disprove such a hypothesis.

In addition to the exploration functions mentioned above, combinations of those exploration functions, such as combining *ϵ-greedy with weight stochastic* or *ϵ-greedy with weighted deterministic*, were also experimented with in the training of the Q-agent.

3.8. State Distillation

Yet another way to reduce the size of the Q-table and improve performance is a novel technique introduced in this thesis known as *state distillation*. The idea behind state distillation is to take two or more states and distill them down to one state, effectively reducing the state space and consequently, the size of the Q-table. In the case of the Pong game, knowing the environment constraint that the speed of the paddles and ball in the y-axis are the same, suppose we want to create a Q-agent whose optimal policy is to follow the ball along the y-axis. Then, the agent would have to have a POMDP Q-table that takes into account at least three observable variables of the Pong environment: 1) the y-coordinate of the ball, 2) the y-coordinate of the paddle, and 3) the action of the paddle. Taking into account these states, the dimensions of the Q-table would be 1000 x 1000 x 3 or 3 million different Q-values.

Now, with enough trials, the Q-agent will eventually learn the optimal policy of the Pong environment. However, 3 million Q-values is a lot to learn and if the agent misses learning even one Q-value properly, it risks diverging from the optimal policy. Fortunately, there is a way to distill the ball y-value and paddle y-value states into one state, as shown in Figure 6. Given that the behavior of the optimal policy involves

minimizing the absolute y-difference between the ball's y-value center and the paddle's y-value center, the distilled state could involve the absolute difference between the ball y-value and paddle y-value states. If the ball's y-value center is within range of the paddle's y-value center, the distilled state is 0; if the ball's y-value center is less than the paddle's y-value center, the state is 1; and if the ball's y-value center is greater than the paddle's y-value center, the state is 2. In effect, we have taken a state space of size one million (1000 x 1000) and distilled it down to just 3 states (0, 1, 2). What's more, these newly distilled states correspond well with the 3 actions of the Pong agent (idle, up, down). By combining this distilled state space with the binary reward system mentioned in Section 3.8, we effectively achieve state distillation of the state space. The resulting distilled state Q-table is of dimension 3 x 3 or just 9 different Q-values! Compared to the size of the undistilled Q-table, this is a 333,333x improvement (3,000,000 / 9 ≈ 333,333) in memory usage! Since the Q-agent has much fewer Q-values to learn, one can expect the Q-learning to be much faster, and the results shown in Section 4 seem promising.

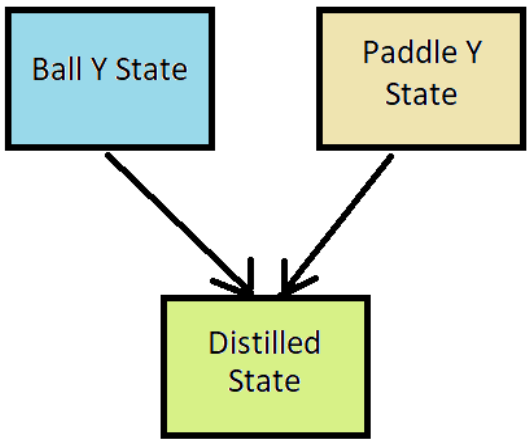


Figure 6: Distilling Pong states into one state

3.9. Q-Learning Design Summary and Conclusion

This section presented a few design considerations which were implemented into the Pong environments over the course of the research. These are listed in Table 4. We began by experimenting with a simple Q-learning implementation in a maze world environment to show the proof-of-concept for Q-learning. Then, the Pong environment's attributes and constraints were discussed. We started with a JavaScript implementation of the Pong environment and then moved to a Python implementation for reasons discussed in Section 3.3. Then, we showed how we can convert an MDP to a POMDP so as to reduce the size of the Q-table. We also went over the Pong AI's Q-function algorithm, reward systems, and exploration functions. Finally, a new concept termed state distillation was introduced as a potential technique for drastically reducing the size of the state space and thus reducing the size of the Q-table and training time as well. In Section 4, we show the results of the experiments conducted using the implementations and techniques discussed in this section, in particular the results of applying state distillation to the Pong environment.

Subsection Number	**Key Decisions Implemented**
3.6	Implemented 3 reward systems: *binary reward system*, *absolute difference reward system*, and *combined reward system*.
3.7	Implemented 3 exploration functions: ϵ*-greedy*, *weighted stochastic exploration*, and *weighted deterministic exploration*.
3.8	Used *state distillation* to convert a Q-table of dimensions 1000 x 1000 x 3 into a Q-table of dimensions 3 x 3.

Table 4: Key decisions made in subsections 3.6 - 3.8

4. Results

This section presents the results of the research. We start by discussing the training procedure used in the experiments in subsection 4.1. The performance metrics used to evaluate these experiments are discussed in subsection 4.2. Then, subsections 4.3 to 4.7 are used to compare the results of the experiments.

4.1. Training Procedure

Prior to each training session, the hyperparameters (α, γ, ϵ, number of trials, number of minimum visits per state, etc.) of the Q-learning are adjusted, the type of reward system is chosen (binary, absolute difference, combined), the type of exploration function is chosen (greedy, ϵ-greedy, deterministic, stochastic, ϵ-greedy + deterministic, ϵ-greedy + stochastic), and the dimensions of the Q-table are adjusted. For training sessions with fewer than 100,000 trials, the Q-learning is done on a local machine and for sessions greater than or equal to 100,000 trials, the learning is done on a remote server and the results (Q-table `.dat` file, output file, graphs) are downloaded to the local machine upon completion of the training. The hardware specs of the server are: two AMD EPYC 7452 CPUs (64 cores), 143 GB memory, and a RTX Quadro 8000 graphic card.

During training of the Q-learning agent, a few changes are made to the Pong environment to aid in faster learning: (1) the Pong GUI is not rendered as it takes extra time to render as shown in Table 3 and is not necessary to play the Pong game, (2) at the

start of each Pong rally, the ball alternates between initialing at the top and the bottom of the canvas so that the Q-agent does not develop a preference or bias to any one side of the canvas, and (3) at every new rally, the Q-agent's paddle, rather than always initializing in the middle of its field, is randomly initialized in its different possible y-positions so that the Q-agent explores all state-action pairs in its Pong environment faster.

After training comes the Pong gameplay, where the Q-agent stops learning and instead uses what it learned during training to play and exploit its environment. The purpose of this gameplay mode is to test the learning of the Q-agent and confirm that it has converged to the optimal policy of the Pong environment. We say the agent has achieved the optimal policy if it manages to consistently win against its opponent with a score of 11 to 0. Changes that were made to aid training are reverted in the actual gameplay. Another key change that is made in gameplay is that the opponent AI is changed from a perfect agent to a near-perfect agent. That is, rather than having a 100% chance of hitting the ball back to the Q-agent, the opponent's paddle only has an 85% chance of hitting the ball upon contact with the ball and, consequently, a 15% chance of missing the ball. This is to prevent the Pong game from running forever, with the ball being indefinitely hit back and forth between the two paddles.

4.2. Performance Metrics

Four main kinds of performance metrics are used to evaluate the performance of Pong AI Q-learning training sessions:

1) The amount of time taken to train the Q-agent.
2) The number of trials per training session.
3) Percent of the state space the Q-agent visited.
4) Whether the Q-agent managed to converge to the optimal policy or not. As previously mentioned in subsection 4.1, the agent is said to have converged to the optimal policy if it manages to consistently win against its opponent with a score of 11 to 0.

While additional performance metrics such as CPU usage, memory usage, etc. could have been considered, in the interest of time they were not used.

4.3 Optimal Policy Training Sessions

In the following three training sections, the Q-agent converged to the optimal policy of the Pong environment. The attributes for these training sessions were chosen based on results found in subsections 4.4 to 4.7.

Attributes of Training Session 1: $\gamma = 0.8$, starting $\epsilon = 1$, ϵ-greedy exploration combined with stochastic exploration, 3 minimum visits per state, no minimum limit on ϵ-decay, absolute difference reward system, and POMDP Q-table of dimension 1000 x 1000 x 3.

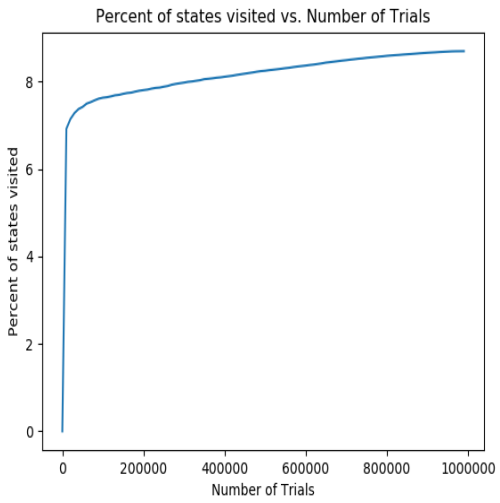
Attributes of Training Session 2: $\gamma = 0.8$, starting $\epsilon = 1$, ϵ-greedy exploration combined with stochastic exploration, 3 minimum visits per state, minimum limit of 0.01

on ϵ-decay, absolute difference reward system, and POMDP Q-table of dimension 1000 x 1000 x 3.

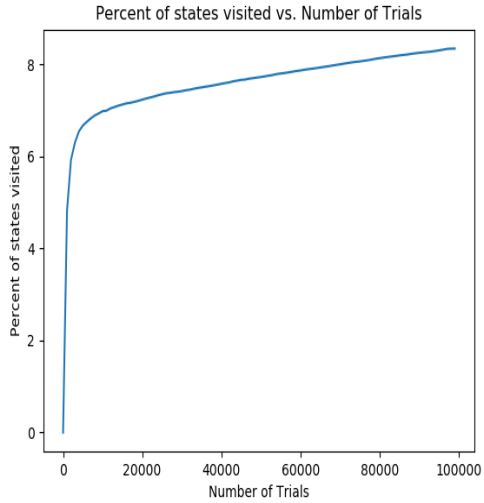
Attributes of Training Session 3: γ = 0.8, starting ϵ = 1, ϵ-greedy exploration, minimum limit of 0.1 on e-decay, binary reward system, and distilled state Q-table of dimension 3 x 3.

Training Session Number	Number of Trials Taken to Train	Time Taken to Train	Percent of State Space Visited	Optimal Policy Achieved?
TS 1	1,000,000	72.65 hrs	8.25%	Yes
TS 2	100,000	8.12 hrs	8.2%	Yes
TS 3	3	6.32 ms	100%	Yes

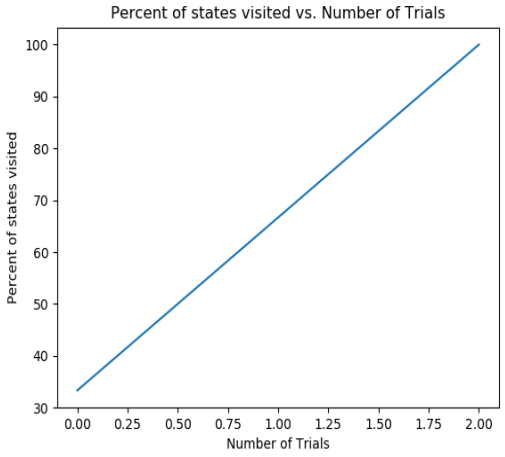
Table 5: Comparison of optimal policy training sessions



(a) TS 1



(b) TS 2



(c) TS 3

Figure 7: Percent of states visited by optimal policy training sessions

From Table 5, we see that Training Session 1 (TS 1) took the most number of trials and the longest time to train the Pong Q-agent. Despite taking almost 9 times as much time as TS 2, the difference in the percent of states visited is very small. Note here that the reason the percent of states visited by these trainings does not reach 100% is because most of the ball y-value and paddle y-value state combinations are physically impossible to reach due to the fact that the ball and paddle move at fixed speeds in the y-axis. So, if we were to only consider the valid Pong states, that would amount to about 8.25% of the state space which is the percent of states visited by TS 1. The most surprising result in Table 5 is TS 3 which uses the state distillation technique introduced in Section 3.10. Using as little as 3 trials, the Q-agent manages to converge to the same optimal policy as TS 1 and TS 2 with a fraction of the time. For a quick time efficiency comparison, TS 3 is 4.6 million times faster than TS 2 and 41.3 million times faster than

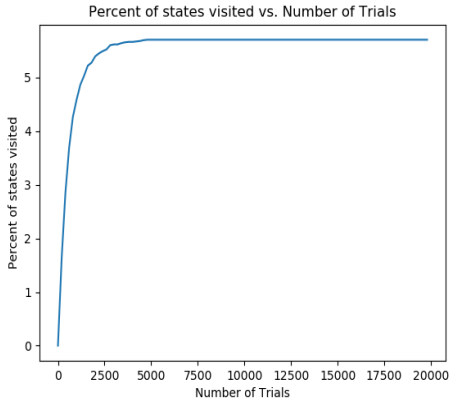
TS 1. Another interesting observation is that while the visited graphs for TS 1 and TS 2 in Figure 7 seem logarithmic, the visited graph of TS 3 is linear. However, this is probably because TS 3 only has 3 distilled states to explore.

4.4. Varying Starting ϵ Values

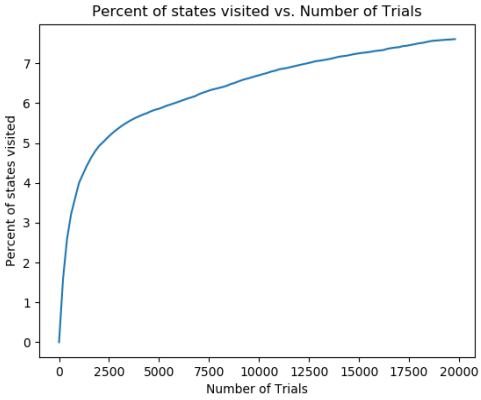
In the following six training sessions, the starting value of ϵ is changed and all other Q-learning hyperparameters are held constant. All of the training sessions are run for 20,000 trials and excluding ϵ, their other attributes are: $\gamma = 0.8$, ϵ-greedy exploration combined with stochastic exploration, 3 minimum visits per state, minimum limit of 0.1 on ϵ-decay (except for TS 4 and TS 9 which both have a minimum ϵ value of 0), absolute difference reward system, and POMDP Q-table of dimension 1000 x 1000 x 3.

TS Number	Initial ϵ	Minimum ϵ Value	Time Taken to Train	Percent of State Space Visited	Optimal Policy Achieved?
TS 4	0	0	26.26s	5.75%	No
TS 5	0.1	0.1	41.84s	7.66%	No
TS 6	0.5	0.1	51.79s	7.66%	No
TS 7	0.8	0.1	53.03s	7.66%	No
TS 8	1	0.1	53.05s	7.75%	No
TS 9	1	0	58.31s	7.75%	No

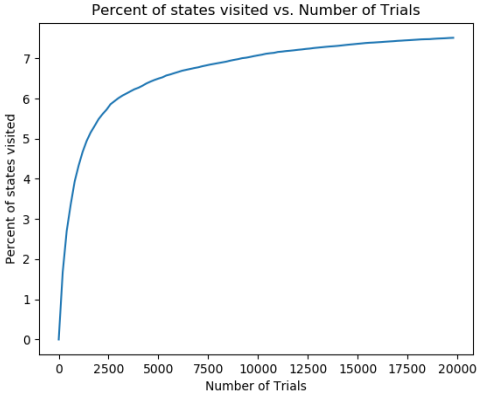
Table 6: Effect of initial ϵ values on training



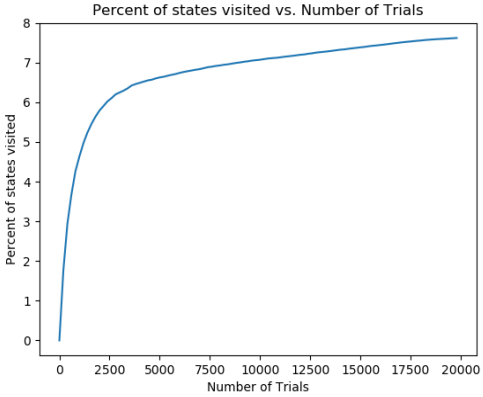
(a) TS 4



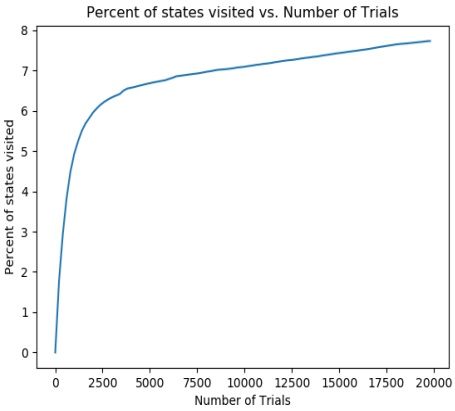
(b) TS 5

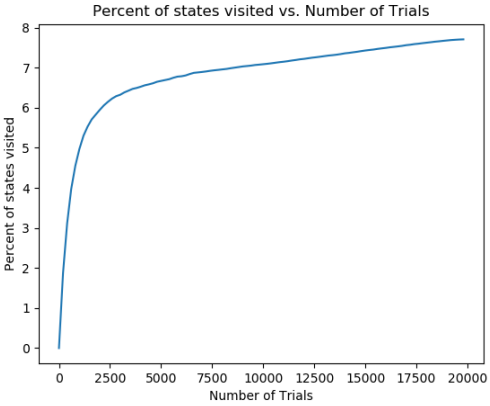


(c) TS 6



(d) TS 7





(e) TS 8 (f) TS 9

Figure 8: Effect of initial and min ϵ values on percent of states visited

From Table 6, as the starting ϵ value (greedy value) increases from 0 to 1, the trend seems to be an increase in training time. Considering that an ϵ value of 0 means the agent focuses solely on exploitation and an ϵ value of 1 means the agent focuses only on exploration of its environment through random actions, the higher the initial ϵ value of the Q-agent, the better chance it has of exploring its environment more thoroughly and learning the optimal policy, hence the increase in the percent of the environment's states visited as ϵ is increased. On the other hand, the downside of increasing the initial ϵ value is greater time spent exploring the environment, resulting in an increase in training time. Another thing to note is that by limiting the ϵ decay to a certain minimum value (like 0.1), the agent is able to finish its training faster, and this is seen by the minimum ϵ values and training times of TS 8 and TS 9. TS 8, which had a minimum ϵ value of 0.1, managed to finish 5 seconds faster than TS 9, which had no limit on ϵ-decay. This is because after completing enough trials, the Q-agent has become mostly competent at interacting in its Pong environment and exploiting it, meaning the agent makes less mistakes and as a result, trials take longer to complete. So, by limiting ϵ to a minimum value and preventing it from decaying all the way to 0, the agent is forced to make mistakes and finish the training trials faster. One last thing to note is that in Figure 8, while the training sessions with initial $\epsilon > 0$ resulted in at least 7.66% of the state space visited, TS 4 which had an initial ϵ of 0 only visited 5.75% of the state space. Without

intentionally exploring its environment using an exploration function, the Q-agent still manages to explore part of the state space by focusing on exploitation alone.

4.5. Varying Discount Factors

In the following five training sessions, only the value of the discount factor γ is changed and all other hyperparameters are held constant. All of the training sessions are run for 20,000 trials and excluding γ, their other attributes are: starting $\epsilon = 1$, ϵ-greedy exploration combined with stochastic exploration, 3 minimum visits per state, minimum limit of 0.1 on ϵ-decay, absolute difference reward system, and POMDP Q-table of dimension 1000 x 1000 x 3.

Discount Factor	Time Taken to Train	Percent of State Space Visited	Optimal Policy Achieved?
0	55.27s	7.66%	No
0.1	56.09s	7.66%	No
0.5	50.94s	7.66%	No
0.8	43.69s	7.66%	No
1	42.78s	7.66%	No

Table 7: Effect of discount factor on training

From Table 7, as the discount factor increases from 0 to 1, the general trend seems to be a decrease in the training time. This makes sense as a higher discount factor means it is easier for future rewards of future state-action pairs to backpropagate to state-

action pairs further back in the state history that receive little to no immediate rewards. Despite higher discount factors improving training times, surprisingly, changing the discount factor has no effect on the percent of the state space visited.

4.6. Efficacy of Exploration Functions

In the following five training sessions, only the exploration function is changed and all Q-learning hyperparameters are held constant. All of the training sessions are run for 20,000 trials and excluding γ, their other attributes are: starting $\epsilon = 1$, ϵ-greedy exploration combined with stochastic exploration, 3 minimum visits per state, minimum limit of 0.1 on ϵ-decay, absolute difference reward system, and POMDP Q-table of dimension 1000 x 1000 x 3.

Exploration Function	Time Taken to Train	Percent of State Space Visited	Optimal Policy Achieved?
ϵ-greedy	43.34s	7.75%	No
weighted stochastic	45.81s	7.15%	No
weighted deterministic	39.32s	5.75%	No
ϵ-greedy + weighted stochastic	58.12s	7.75%	No
ϵ-greedy + weighted deterministic	41.67s	5.75%	No

Table 8: Effect of various exploration functions on training

From Table 8, the two exploration functions with the highest percent of states visited are the *ϵ-greedy* and *ϵ-greedy plus weighted stochastic* exploration functions. Of these two, *ϵ-greedy plus weighted stochastic* lasted the longest, indicating that the agent stays alive in the Pong environment longer. The reason weight deterministic exploration has the smallest percentage of states visited is because under this exploration, the Q-agent prematurely stops exploring its environment and goes into exploitation mode, converging the policy to a bad local optima. Since the Q-agent makes more bad decisions under this exploration function compared to the others, the agent lasts for a shorter amount of time in the environment, and so the training takes the least amount of time. One important thing to note is that of these exploration functions, only *ϵ-greedy plus weighted stochastic exploration* consistently resulted in the Q-agent converging to the optimal policy.

4.7. Other Results

In addition to the results presented in the subsections above, there are two more results that are of interest. The first result is the number of trials completed over a time span of 6 hours of a 100,000-trial training session. In Figure 9, we see that the graph resembles a log curve. Out of the 100,000 total trials in the training session, at least 99,000 of the trials are completed within the first 15 minutes of training. This means that most of the time (about 5.75 hours) is spent in the last 1,000 trials training the Q-agent to converge completely to the optimal policy.

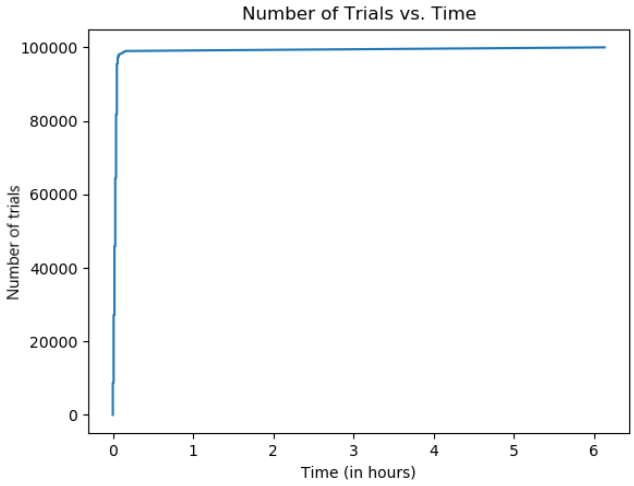


Figure 9: Number of trials vs. Time

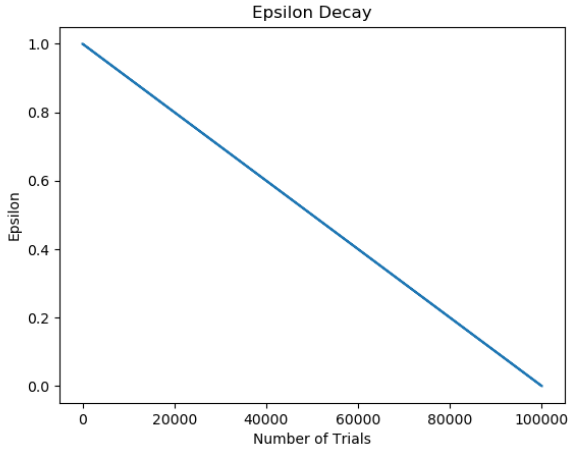


Figure 10: Epsilon decay for 100,000 trials

The second result is the decreasing value of epsilon, ϵ, in ϵ-greedy exploration as the number of trials increases. From Figure 10, we can clearly see that the ϵ value decreases linearly with the increase in the number of trials completed. What is interesting to note from these last two results is that by using a linearly decreasing ϵ for ϵ-decay, the

competence of the Q-agent increases logarithmically. Considering that a lower ϵ value means greater exploitation of the environment by the agent, these results suggest that there is diminishing returns to decreasing ϵ beyond a certain ϵ value (e.g., $\epsilon = 0.1$). For this reason, it makes sense to place a minimum limit on the decay of ϵ. In doing so, training times will be reduced and the Q-agent will have mostly converged to the optimal policy.

The main idea behind presenting the results in subsections 4.4 to 4.7 is to highlight the experiments that were conducted to choose the best attributes to use in the training sessions which yielded optimal policies in subsection 4.3. Of the three training sessions presented in subsection 4.3, TS 3, which combined the use of a POMDP Q-table with state distillation, proved to be the most performant. Using only three trials and taking 6.32 ms to run, TS 3 far outcompetes its competition, TS 1 and TS 2, in terms of number of trails required and runtime.

5. Related Work

5.1. Learning to Utilize Shaping Rewards

As was previously mentioned in Section 3.8, reward shaping is a way of improving the learning rate by introducing background knowledge into model-free reinforcement learning algorithms like Q-learning (Asmuth 2008). The term shaping originates from the animal-learning literature. “In training animals, shaping is the idea of directly rewarding behaviors that are needed for the main task being learned” (Asmuth 2008). In the RL community, “early work of reward shaping can be dated back to the attempt of using hand-crafted reward function for robot behavior learning and bicycle driving” (Hu 2020). For many learning problems, the goal can often be broken down into smaller and simpler subgoals. By using pseudorewards to encourage the RL agent to achieve the subgoals, one can put it on a better, faster track to achieving the main goal compared to an agent trained only on the sparse rewards it receives from its environment. In a sense, we convert domain knowledge of the environment into pseudorewards in the reward function to provide hints about the optimal behavior to the agent. As long as adding pseudorewards does not alter the optimal policy, given enough training, the agent will converge to the optimal policy as dictated by the shaping theorem (Krueger 2018).

While reward shaping is a powerful technique for supplementing and accelerating an agent’s learning, it has one major caveat: sometimes the domain knowledge may be unknown, unclear, or only partially known. In such instances, it can be hard to determine which pseudorewards or what amount of those pseudorewards should be used to reward

which subgoals so as to get the agent to learn the optimal policy. There is no clear way of knowing ahead of time which pseudorewards will prove to be the most beneficial for learning, and so a lot of experimentation and guessing is involved in the process.

The research paper titled *Learning to Utilize Shaping Rewards: A New Approach of Reward Shaping* by Yujing Hu et al. (2020) addresses this problem by adaptively utilizing a given shaping reward function. In the paper, the utilization of shaping rewards is formulated as a bi-level optimization of parameterized reward shaping (BiPaRS) problem. The lower level optimizes the policy using the shaping rewards while the upper level optimizes a parameterized shaping weight function. Combining these two levels makes it possible to achieve true reward maximization. The gradient of the expected true reward is derived with respect to the shaping weight function parameters. While the gradient cannot be directly computed, it is instead approximated. Three algorithms are used for the gradient approximation: 1) Explicit Mapping (EM), 2) Meta-Gradient Learning (MGL), and 3) Incremental Meta-Gradient Learning (IMGL). Based on the results from the experiments conducted in the cartpole and MuJoCo environments, it has been shown that the algorithms "can fully exploit beneficial shaping rewards, and meanwhile ignore unbeneficial shaping rewards or even transform them into beneficial ones" (Hu 2020). The results found in this paper serve as motivation for a potential future implementation of the Pong AI reward system that helps the Q-agent learn the optimal policy faster than current handcrafted pseudorewards.

5.2. Q-learning Convergence for Non-Markov Decision Processes

In Section 3.6, the MDP of the Pong environment was converted to a POMDP for the purposes of creating a smaller Q-table, using less memory, and making the Q-learning feasible. It was also mentioned in that section that due to known environmental constraints, the POMDP can actually be treated as a regular MDP. Likewise, in Section 2.3, it was briefly mentioned that TD Q-learning has been shown to converge well on some kinds of non-MDP problems. This makes one wonder what classes of non-MDP problems TD Q-learning converges in and what classes it does not.

A paper titled *On Q-learning Convergence for Non-Markov Decision Processes* by Sultan Javed Majeed and Marcus Hutter (2018) aims to answer that question. Owing to the fact that most real-world problems are inherently non-Markovian, the authors of the paper investigate the behavior of TD Q-learning in non-MDP and non-ergodic domains with potentially infinite underlying states. Decision processes are classified into 4 types: 1) MDP, 2) POMDP, 3) History-based Decision Process (HDP), and 4) Q-value Uniform Decision Process (QDP). Definitions for MDP and POMDP can be found in Section 2.1. The HDP is "a stochastic mapping from a history-action pair to observation-reward pairs" (Majeed 2018), or basically any decision problem where it is necessary to keep track of the state history in order to make an informed decision in the current state. QDP is a class of environments in which for any action a, if there exist any two histories h and $\dot{h}$ which map to the same state s, then the optimal Q-values of the underlying HDP of these histories are the same and the state-uniformity condition of the QDP is met. The QDP class is the class of problems that can be solved by Q-learning. It completely

encompasses the MDP class and has a non-empty intersection with POMDP and HDP as illustrated in Figure 11.

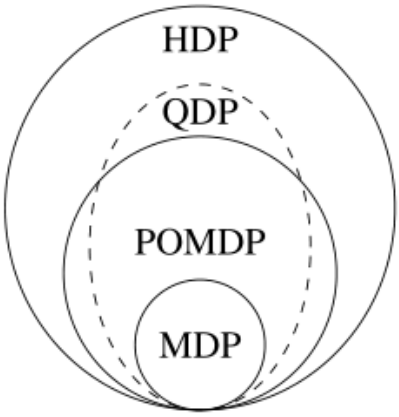


Figure 11: Euler diagram for MDP, POMDP, HDP, and QDP

In the paper, a convergence proof is provided proving that Q-learning converges in all QDP class problems. In addition, it has been shown that the "state-uniformity of the optimal Q-value function is a necessary and sufficient condition for Q-learning to converge even in the case of infinitely many internal states" (Majeed 2018). The implication of this proof is that while Q-learning can solve all MDP problems, it can only solve some POMDP and HDP problems. So in the event that the Pong AI game is made in such a way that it resembles a non-stationary (joint probability distribution of process changes over time) POMDP or HDP problem, there is a chance that such a RL problem could be solved using TD Q-learning, which is impressive considering that finite-state MDPs are only able to model stationary domains. In the future, similar to the convergence proof for QDP, a convergence proof for one-step Q-learning can be made to showcase the classes of decision problems it can or cannot solve.

5.3. Playing Atari with Deep Q-Learning

The idea of playing Pong using Q-learning draws inspiration from a research paper published under DeepMind Technologies titled *Playing Atari with Deep Reinforcement Learning* by Volodymyr Mnih et al. (2013). Unlike this thesis which uses one-step Q-learning alone to perform the learning in the Pong environment, their paper combines one-step Q-learning with a convolutional neural network to play not just Pong but six other Atari games as well (Holcomb 2018).

Artificial neural networks are powerful learning architectures that try to mimic the neural interconnections and communications that happen in the human brain. The networks are arranged into three main types of neural layers: 1) input layer, 2) hidden layer, and 3) output layer. Inputs to the network are fed into the input layer and network's outputs come out of the output layer. In between the input and output layers are the hidden layers. There can only be one input and output layer but many hidden layers. The outputs from each layer are fed into the next layer. In a fully connected network, each neuron in each layer is connected to every neuron in the next layer as illustrated in Figure 12 (Kilgannon 2020). Each connection has a weight associated with it and this weight can be increased or decreased to strengthen or weaken connections between the neurons.

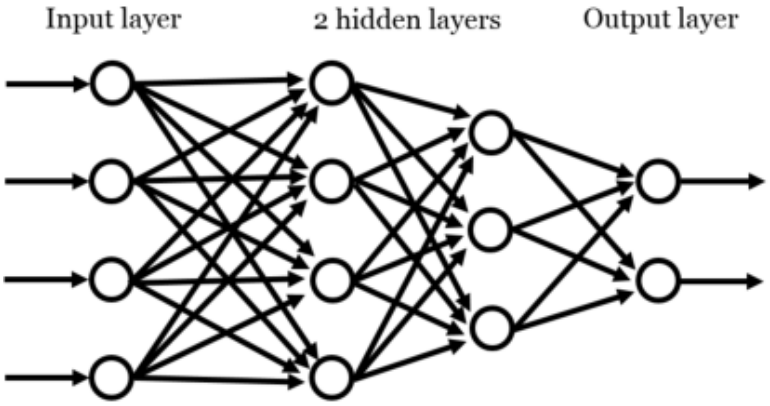


Figure 12: Fully connected neural network design

According to the universal approximation theorem, a network with more than one layer can approximate any continuous function to "an arbitrary degree of accuracy" (Russell 2021). Training a neural network involves feeding inputs forward to the network and adjusting the weights of the network based on the accuracy of the network's predictions to the actual results in a process known as backpropagation. After many forward and backward passes of training examples through the network (also known as epochs), the network will have approximated the intended function (Russell 2021).

Since neural networks essentially act as function approximators, they can be used to approximate the Q-function ($Q(s, a; \theta) \approx Q*(s, a)$). The neural network function approximator with weights θ is referred to as a Deep Q-network (DQN) (Mnih 2013). The loss function, the function which highlights the error between the predicted and actual Q-values is defined as follows:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s, a; \theta_i))^2]$$

Equation 7: The Q-network MSE loss function

The loss function depicted in Equation 7 is known as the Mean Squared Error (MSE) loss function (Brownlee 2020). The error is measured as the difference between the actual Q-value y_i and the network's estimated Q-value. All of the network's errors are squared and then averaged to get the MSE. By reducing the value of the loss using backpropagation via a stochastic gradient descent optimizer, the network will more accurately approximate the Q-function. The optimizer algorithm will usually contain a learning rate, and this learning rate is typically set between 0.001 and 0.01 in order to promote a fast decrease in the loss of the loss function (Surmenok 2017). The benefit of using a DQN over a regular Q-function is that due to the nature of neural networks, the DQN is able to generalize between state-action pairs, so unlike regular Q-learning, not every state-action pair in the environment has to be visited multiple times for the agent to converge to the optimal policy (Mnih 2013). While a feedforward network is enough to learn the Q-function, it does not work well when the inputs are images instead of state attributes. For image inputs, Convolutional Neural Networks are used.

Convolutional Neural Networks (CNNs) were developed from traditional neural networks and were inspired by an animal's visual cortex's ability to retain features from an image or images (Li 2020). Like the visual cortex, a CNN retains relationships between pixels in 2D or 3D space using feature maps generated by moving a kernel filter

stepwise across the input image. Generating feature maps in this way helps to "decrease the memory footprint of the network by reducing the number of weights in the network using weight sharing among kernel filters" (Kilgannon 2020). So in the case of a Pong game image, the CNN will be able to create feature maps to make out the features of the paddles, the ball, and with the use of multiple images, even the movements of the paddles and the ball. This is exactly how it is implemented in Mnih's paper. Batches of *k* input frames (where *k* = 4 or 3) are fed into a CNN to train the DQN to play Pong and other Atari games. As is characteristic of the one-step Q-learning shown in Equation 4, rewards are combined with the discounted max future Q-values to get the updated Q-values and these new Q-values in turn are used to correct the Q-value approximations outputted by the network. The pseudocode for the DQN algorithm is presented in Mnih's paper as follows:

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Algorithm 3: DeepMind Technologies' DQN Algorithm

Ignoring the details of the DQN algorithm, one of the things that stand out in Algorithm 3 is the use of an experience replay buffer, from which previous experiences are randomly sampled from and fed to the network (Zychlinski 2019). This buffer exists to serve three main functions. First, compared to supervised learning, data is scarce in RL applications, so by replaying previous experiences, the data collected is used efficiently. Second, random experience sampling reduces strong correlations between consecutive samples and therefore reduces the variance of the updates. Third, by using experience replay, "the behavior distribution [of the agent] is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters" (Mnih 2013). To ensure the experience replay buffer does not grow indefinitely and consume all of the system's memory, only the last N experience tuples are stored in the buffer. Note that the reason experience replay works with DQNs is that Q-learning is an off-policy algorithm. For on-policy algorithms, experience replay does not work.

The results of applying the DQN algorithm on Atari games were quite impressive. It was found to have outperformed all previous RL approaches on six out of the seven games experimented on and even managed to surpass a human expert on three of them (Mnih 2013). Recently, another kind of network called Recurrent Neural Network (RNN) was combined with DQN to form a new kind of network called Deep Recurrent Q-Network (DRQN) (Upadhyay 2019). Since RNNs are typically used for ordinal or temporal problems, they are able to take advantage of certain temporal aspects of state sequences (such as a ball moving in between frames) which DQNs oftentimes neglect as they have limited or no amount of distant history (Brownlee 2019). The recurrent layers in a DRQN

are used to extract meaningful signals out of the sequence of features extracted from images using convolutional layers in the DRQN. From experiments conducted, DRQNs have been shown to perform better than DQNs.

5.4. OpenAI Gym Pong Environment

Initially, when thinking of ways to implement a Pong GUI in Python, we came across a Pong environment in OpenAI Gym (see Figure 13). “Gym is an open-source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API” (“Gym” 2021). It was created to satisfy the RL research community’s need for good, standardized benchmarks on which to compare their algorithms. The library is convenient, accessible to use, and contains a diverse collection of environments with a common interface (Brockman 2016). One of these environments is an Atari 2600 Pong environment called Pong-v0. “In this environment, the observation is an RGB image of the screen, which is an array of shape (210, 160, 3). Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from {2,3,4}” (“Gym” 2021). In each iteration of the game, the agent chooses an action to take and the environment gets updated based on that action. After updating, the environment returns the observation of the environment as an image array, a reward as an integer, a boolean indicating whether

the game is done or not, and other info. In this way, the agent interacts and learns from its environment.

However, the Gym's Pong environment is restrictive. While Gym's environments are great for standardization and for comparing algorithms, it does not allow the user to alter its environments in any way. While it is possible to modify and change the observation array, it is not possible to modify the environment itself, such as to increase the size or speed of the paddles. To do such things, one would need to create their own environment.

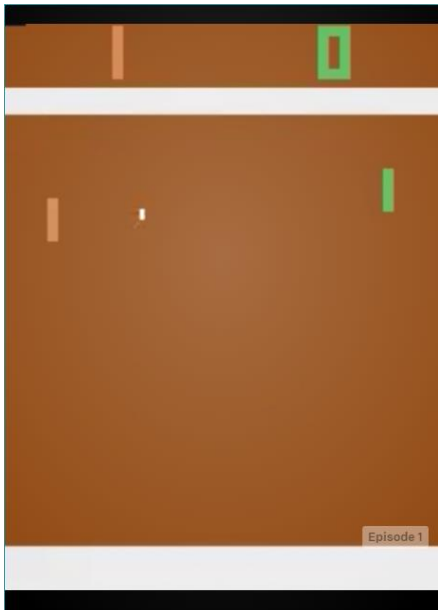


Figure 13: OpenAI Gym Pong-v0 environment

6. Future Work

Much experimentation remains to be conducted with the state distillation technique. Being a new technique, there is so much left to explore with regards to the potential and limitations of distilling states. If a large state space can be reduced to a smaller state space by distilling its states into more meaningful distilled states, one has to wonder whether the distilled states themselves can be further distilled into fewer concentrated states. This way of recursive state distillation is not unlike the idea of extracting high-level feature maps from low-level features in CNNs.

In the Pong environment, state distillation proved to greatly reduce the size of the discrete Pong state space and thus speed up learning. While this shows that state distillation on discrete state spaces can be used beneficially, we cannot say for sure that it will work on all kinds of discrete state spaces or even on continuous state spaces. To make such guarantees, a formal proof of state distillation is needed, and this is something that is planned to be done in the future.

Finally, it would be interesting and valuable to see if the benefits of state distillation can be applied to other kinds of reinforcement learning algorithms besides one-step Q-learning, in particular with regards to DQNs. Furthermore, experiments in deep learning comparing the effects of different types of Q-learning, different kinds of network layers, number of layers, number of neurons per layer, loss functions, etc. on deep Q-learning may also be conducted.

7. Conclusion

In this thesis, we explored a few ways to make the training of the Pong Q-agent more efficient. By considering the MDP of the Pong environment as a POMDP and treating the POMDP as a regular MDP due to known environmental constraints, the Pong state space was able to be reduced by a factor of 5.6 million, resulting in a smaller Q-table with 3 dimensions instead of 7. Then, by combining the ball y-value and paddle y-value states into one distilled state using the novel state distillation method, the size of the state space was further reduced by a factor of 333,333, resulting in a Q-table of just 9 Q-values. From experiments conducted, it is clear that reducing the Q-table in this way greatly helps to reduce the number of trials required and thus the time required to train the Q-agent. While previous attempts at training RL agents to play Pong involved using complex deep neural networks combined with Q-learning (DQNs), this thesis highlights that it is possible to use one-step Q-learning, a model-free learning algorithm typically relegated to solving simple maze world environments (see Section 3.1), in combination with POMDPs and state distillation to train a Q-agent to play Pong and converge to the optimal policy.

References

"Gym: A Toolkit for Developing and Comparing Reinforcement Learning Algorithms". *Gym OpenAI*, 2021, https://gym.openai.com/.

"Introducing Asynchronous Javascript - Learn Web Development | MDN". *MDN Web Docs*, 2021, https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing.

"Pong Game". *Pong Game.org*, 2021, https://www.ponggame.org/.

"Pyglet Documentation — Pyglet V1.5.21". *Pyglet Read the Docs*, 2021, https://pyglet.readthedocs.io/en/latest/.

Arora, Saurabh, and Prashant Doshi. "A Survey of Inverse Reinforcement Learning: Challenges, Methods and Progress" *Artificial Intelligence* (2021): 103500.

Asmuth, John, Michael L. Littman, and Robert Zinkov. "Potential-Based Shaping in Model-Based Reinforcement Learning" *AAAI*. 2008.

Bellman, Richard. "Dynamic Programming" *Science* 153.3731 (1966): 34-37.

Bhatt, Shweta. "Reinforcement Learning 101". *Medium*, 2018, https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

Blackburn. "Introduction to Reinforcement Learning : Markov-Decision Process". *Medium*, 2019, https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da.

Brockman, Greg, et al. "OpenAI Gym" *arXiv preprint arXiv:1606.01540* (2016).

Brownlee, Jason. "How to Choose Loss Functions When Training Deep Learning Neural Networks". *Machine Learning Mastery*, 2020, https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/.

Brownlee, Jason. "When to Use MLP, CNN, And RNN Neural Networks". *Machine Learning Mastery*, 2019, https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/.

Coggan, Melanie. "Exploration and Exploitation in Reinforcement Learning" *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University* (2004).

Hansen, Eric A. "Solving POMDPs by Searching in Policy Space" *arXiv preprint arXiv:1301.7380* (2013).

Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning" *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. No. 1. 2016.

Hasselt, Hado. "Double Q-Learning" *Advances in Neural Information Processing Systems* 23 (2010): 2613-2621.

Holcomb, Sean D., et al. "Overview on DeepMind and Its AlphaGo Zero AI" *Proceedings of the 2018 international conference on big data and education*. 2018.

Hu, Yujing, et al. "Learning to Utilize Shaping Rewards: A New Approach of Reward Shaping" *arXiv preprint arXiv:2011.02669* (2020).

Kilgannon, Jon, "A Machine Learning System for Glaucoma Detection Using Inexpensive Machine Learning" (2020). *West Chester University Master's Theses*. 172. https://digitalcommons.wcupa.edu/all_theses/172

Krueger, Paul, et al. "Shaping Model-Free Reinforcement Learning with Model-Based Pseudorewards" *Proc. Conf. Cogn. Comput. Neurosci*. 2018.

Lee, Dan. "Reinforcement Learning, Part 5: Monte-Carlo And Temporal-Difference Learning". *Medium*, 2019, https://medium.com/ai%C2%B3-theory-practice-business/reinforcement-learning-part-5-monte-carlo-and-temporal-difference-learning-889053aba07d.

Lee, Dan. "Reinforcement Learning, Part 6: TD(λ) & Q-Learning". *Medium*, 2019, https://medium.com/ai%C2%B3-theory-practice-business/reinforcement-learning-part-6-td-%CE%BB-q-learning-99cdfdf4e76a.

Li, F., Ranjay Krishna, Danfei Xu, and Amelie Byun. "CS231n: Convolutional Neural Networks for Visual Recognition" (2020) Stanford University. Web. http://cs231n.stanford.edu/

Majeed, Sultan Javed, and Marcus Hutter. "On Q-Learning Convergence for Non-Markov Decision Processes" *IJCAI*. 2018.

Mnih, Volodymyr, et al. "Playing Atari with Deep Reinforcement Learning" *arXiv preprint arXiv:1312.5602* (2013).

Modany, Angela. "Pong, Atari, and the Origins of the Home Video Game". *National Museum of American History*, 2012, https://americanhistory.si.edu/blog/2012/04/pong-atari-and-the-origins-of-the-home-video-game.html.

Ng, Andrew Y. *Shaping and Policy Search in Reinforcement Learning*. University of California, Berkeley, 2003.

Peng, Jing, and Ronald J. Williams. "Incremental Multi-Step Q-learning" *Machine Learning Proceedings 1994*. Morgan Kaufmann, 1994. 226-232.

Rocca, Joseph. "The Exploration-Exploitation Trade-Off: Intuitions and Strategies". *Medium*, 2021, https://towardsdatascience.com/the-exploration-exploitation-dilemma-f5622fbe1e82.

Russell, Stuart J et al. *Artificial Intelligence: A Modern Approach*. 4th ed., Pearson Education, 2021.

Shinners, Pete. *Pygame.org*, 2000, https://www.pygame.org/.

Shyalika, Chathurangi. "A Beginners Guide to Q-Learning". *Medium*, 2019, https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c.

Surmenok, Pavel. "Estimating an Optimal Learning Rate for a Deep Neural Network". *Medium*, 2017, https://towardsdatascience.com/estimating-optimal-learning-rate-for-a-deep-neural-network-ce32f2556ce0.

Torres, Jordi. "The Bellman Equation". *Medium*, 2020, https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7.

Upadhyay, Uddeshya, et al. "Transformer Based Reinforcement Learning for Games" *arXiv preprint arXiv:1912.03918* (2019).

Uther, William TB, and Manuela M. Veloso. "Tree Based Discretization for Continuous State Space Reinforcement Learning" *Aaai/iaai* 98 (1998): 769-774.

Vedpathak, Omkar. "Playing Pong Using Reinforcement Learning". *Medium*, 2019, https://towardsdatascience.com/intro-to-reinforcement-learning-pong-92a94aa0f84d.

Watkins, Christopher JCH, and Peter Dayan. "Q-Learning" *Machine learning* 8.3-4 (1992): 279-292.

Zychlinski, Shaked. "The Complete Reinforcement Learning Dictionary". *Medium*, 2019, https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e.

Appendices

The following appendices are included to provide additional detail into the Pong AI Q-learning implementation and its optimizations, including a state reduction technique known as state discretization.

A. The Q-Agent Class

The `Qagent` class is the class that is used to create the Q-agent object. This class is responsible for the actions and decisions of the Q-agent and contains the following methods: an __init__ function for defining the hyperparameters and Q-table for the Q-learning algorithm, a reward function for assigning rewards for state-action pairs, an exploration function for helping the Q-agent to effectively explore its environment, a Q-function for calculating the Q-values of state-action pairs, a Q-learn function for making the Q-agent learn about its environment, a play_game function for making the Q-agent play the Pong game after having learned all the Q-values, and other functions for graphing and debugging purposes.

`Qagent` Function	**Purpose**
__init__(pong_game, paddle, opp, ball)	to define Q-learning hyperparameters and initialize Q-table.
r(s, a)	to give the agent a reward for taking action *a* in state *s*.
exploration_fn(ball_y, pad_y)	to explore unseen states in the environment while reducing the probability of revisiting bad states.
q(s, a)	to calculate Q-values for state-action pairs.

qlearn(ball_y, pad_y)	to make the Q-agent learn about its environment.
play_game(ball_y, pad_y)	to make the Q-agent play the Pong game after having learned all the Q-values.

Table 9: `Qagent` Class Functions

B. Optional GUI Rendering

As was previously demonstrated in Section 3.3, playing the Pong game with the Pyglet GUI excluded resulted in faster training time compared to the GUI being included. However, without the GUI, it can be hard to debug the program and see whether the Q-agent is playing as intended. Ideally, we would like the GUI to be excluded during training and included during demonstration of the learned policy (testing). This is achieved by using a boolean that indicates whether the Q-agent is in Q-learning mode or exploitation mode. When the boolean is set to true, the Q-agent is in Q-learning mode and the Pong GUI is not rendered. The entire game and all its contained objects are manipulated purely through computations. When the boolean is set to false, the Q-agent is in exploitation mode and the Pong GUI is rendered on the monitor. By using this optional GUI rendering approach we get the best of both worlds.

C. Saving the Q-table

As the number of trials required to train the Pong Q-agent increases, it becomes more computationally expensive and time consuming to have to re-train the Q-agent each

time to demonstrate its learning. To avoid this scenario, after completion of each training session, the Q-table is saved to a `.dat` file using pickling in Python. Then later, the Q-table can be loaded from the `.dat` file and used by the Q-agent as if it had been trained on the Pong environment already. This saves a great deal of time and computational resources. In addition, intermediate Q-tables can also be saved in-between the start and end of a training session to show the progress of the Q-learning and/or prevent loss of learning progress.

D. State Discretization

State discretization was incorporated into the Pong game to save time and computation. Reinforcement learning is great for learning policies in discrete state environments, but as the size of the state space increases linearly, its efficiency decays exponentially. “Large state spaces affect learning speed, but often the exact location in that space is not relevant to achieving the goal” (Uther 1998).

In the case of the Pong game, we have a 1400 px by 1000 px canvas. Suppose we were to create a POMDP Q-table that takes into account four observable variables of the Pong environment: 1) the x-coordinate of the ball, 2) the y-coordinate of the ball, 3) the y-coordinate of the Q-agent’s paddle, and 4) the action of the paddle. Then, the dimensions of the Q-table would be 1400 x 1000 x 1000 x 3 or 4.2 billion different Q-values!

Now consider that in this particular Pong environment it does not matter what part of the paddle the ball bounces off of. Whether the ball bounces at top, middle or bottom of the paddle does not affect the speed or angle of the bounced ball. Also consider that the paddles do not go outside the boundaries of the Pong canvas. Given these environmental constraints, we could discretize the state spaces of the paddle and the ball. Without discretization, the paddle has 1000 possible states (some of which are physically unreachable by the paddle due to its speed being greater than 1 pixel). However, if we take into account that the height of the paddle is 70 px, we see that if we line up some paddles along the y-axis of the Pong canvas, we can fit about 14.3 paddles. So, after discretization of the paddle's state space, we end up with 15 possible paddle states (ceiling of 14.3) instead of the previous 1000 states. Similarly, if we arrange copies of the ball in a line along the y-axis of the canvas, we can fit about 55.6 balls. So, after discretizing the ball's y-coordinate state space, we reduce the 1000 possible ball y-value states into 56 ball y-value states. Likewise, the ball's x coordinate can also be discretized and reduced from 1400 ball x states to 78 ball x states. With the newly discretized paddle and ball states, the dimensions of the new Q-table is 78 x 56 x 15 x 3 or over 196,560 different Q-values.

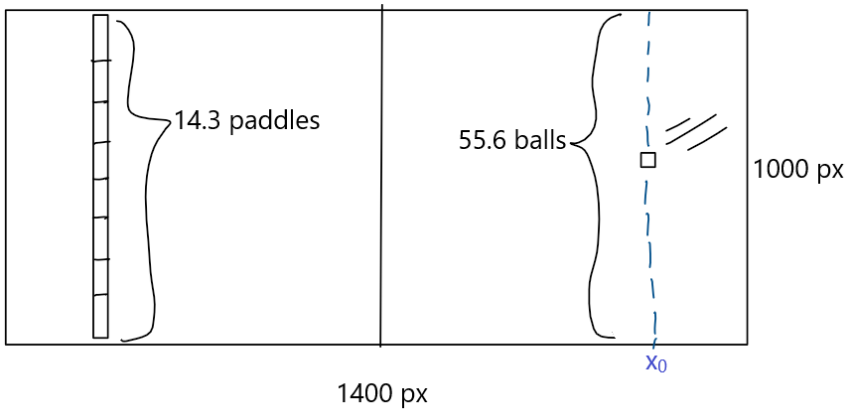


Figure 14: State discretization of the paddle and ball state spaces

Compared to the former Q-table of 4.2 billion Q-values, the discretized Q-table of around 200 thousand Q-values is a 21,367x improvement in memory efficiency. Since there's much less Q-values to learn in the new Q-table, one can expect the Q-agent to train in less time. However, experiments conducted with the discretized Q-table failed to yield good results. That doesn't entail that state discretization is useless, but more so that the code implementation of state discretization was likely incorrect.