PNRL: A system to support Reinforcement Learning development in an undergraduate level AI course

by

Chinmay Arvind

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

B.SC COMPUTER SCIENCE HONORS

in

IRVING K. BARBER SCHOOL OF ARTS AND SCIENCES

Faculty of Science
The University of British Columbia
Okanagan

Abstract

Reinforcement Learning (RL) is a key area of research in Artificial Intelligence (AI), and has been used to create game-based learning environments for training AI agents previously. However, the complexity of these environments has been either too basic or too advanced to be adopted as a tool to teach RL concepts within an undergraduate level AI course. In this thesis, a system called PNRL (Puddle Navigator Reinforcement Learning) is proposed, that uses a client-server architecture to create an environment for game-based learning to take place and support student learning of RL concepts at a sufficiently complex level for undergraduate students in an AI course. The puddle world problem is a foundational RL problem in which an AI agent needs to learn how to navigate a grid world containing some states that produce a negative reward by gaining more experience through its training. The puddle world problem introduces and makes of use of the most important concepts in RL and can be used to teach those concepts to undergraduate students.

This thesis explains the architecture of PNRL, its communication protocol that enable students to write parts of the client-side agent so moves are made and sent to the server, allowing instructors to evaluate the student's understanding of RL concepts. The instructor's evaluation can be guided by assessing the client-side (student's) agent's learning stability in its training process. The system as a whole is tested for its effectiveness in handling complex communication protocols and the client-side agent is tested for its effectiveness in achieving learning stability. A comparison of agent training termination conditions in terms of time-feasibility and learning stability is also conducted to find the most feasible condition in supporting student learning of RL concepts. The agent implemented in PNRL achieves learning stability, the server-side is deemed reliable in handling communications from the client-side, and the maximum steps training termination condition is deemed most feasible to support student learning.

This thesis also briefly introduces a Python to Java code integration for the Game of Amazons project in UBC's COSC 322 AI course.

Preface

This thesis is the original intellectual work of the author, Chinmay Arvind, and the research within this thesis was completed under the supervision of Dr. Yong Gao.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial intelligence (AI) systems are increasingly becoming a part of modern day life and workflows in many subject domains. The need for producing machine learning (ML) systems that learn from data of interest and use that knowledge to take action as organizations see fit have increased. Reinforcement learning (RL) is one specific type of ML technique in which an RL agent makes decisions and takes actions in an environment so as to maximize its reward received from performing different actions. An instance where reinforcement learning can be seen in action is when a robot that can move, pick up trash, and dispose it in a bin makes a decision on whether to keep looking for more trash to dispose of or return to its battery charging station [1]. Reinforcement learning agents have been successfully used in many practical applications such as: playing the Jeopardy! quiz game to a level exceeding human capabilities, managing Dynamic Random Access Memory (DRAM) in computers, achieving human-like ability to play Atari 2600 video games, defeating human experts at the game of Go, and in personalizing web services to significantly increase user engagement with web services [2].

While reinforcement learning has been used extensively in game-related applications, there is a lack of adoption of using construction of reinforcement learning systems in a game-based learning context to support learning of RL concepts at the undergraduate education level. This deprives students of seeing these concepts in action, likely leading to students not being able to understand the concepts fully, and unsatisfied learning outcomes. Game-based learning has been shown to improve students' understanding of programming concepts and motivation provided that there are tangible links from the subject matter to the game itself [6–8].

The introduction of a system that links concepts of AI to games that introduce AI concepts has previously been shown to increase the quality of teaching of these concepts at the K-12 education level [3, 4]. One such example of a game that taught students at the K-12 education level about AI concepts was ArtBot, which was applauded by students and teachers alike for its ease of use for students to choose how their AI agent would play the game[5]. The game logic for a similar system can be designed and extended to include tasks of appropriate difficulty to become a part of undergraduate level curriculum in AI courses. Two examples of more complex reinforcement learning development systems that could be used for

building game-based learning systems are the Arcade Learning Environment (ALE), which is a platform for building and testing reinforcement learning agents to play more decision oriented games such as Atari 2600 games [9], and Project Malmo, a Minecraft-like platform for building and testing agents on various complex tasks in a continuous 3D environment [10].

The puddle world problem [11] is a standard RL problem consisting of a grid world containing puddles that produce a negative reward for a reinforcement learning agent that steps in a puddle state. The objective of the reinforcement learning agent is to learn a strategy to traverse through the puddle world while avoiding the puddles and hence, maximizing the cumulative reward. The puddle world is of size $n \times n$, where n is the number of rows and columns of grid squares that control the size of the world that the agent is traverses within.

The Game of Amazons is a dual player game where players have to move their set of pawns to restrict the other player's pawns from being able to move with the use of arrows that can be shot to block an opponent's path. An extension of the game-based learning approach to benefit undergraduate students would be to allow them to gain practical experience by building parts of a reinforcement learning agent that learns to navigate a puddle world and parts of an AI agent that plays the Game of Amazons to help them gain a deeper understanding of the underlying learning algorithms and when to apply them effectively.

Students coming from different educational and technical backgrounds may have learned different programming languages, and are sometimes restricted to using a single programming language that they may not be familiar with in an undergraduate level programming course. This often requires them to learn syntax of a programming language rather than focus on the logic of the application that they are building, leading to ineffective hands-on learning of the concepts that are behind developing an efficient application. This scenario underscores the importance of creating cross-language compatible game-based learning systems to aid students in focusing on the task at hand rather than syntax of a programming language. As of 2024, Python and Java are ranked as the 4th and 6th most popular programming languages according to a Stack OverFlow survey [12], and they have been used in undergraduate level programming courses for a long period of time, therefore, modifying a previously single programming language game-based learning system to a cross-language compatible system will benefit students immensely.

This thesis introduces a system called PNRL (Puddle Navigator Reinforcement Learning), a client-server-based architecture containing the logic of a game-based RL system. It includes a fully-functional RL agent within a puddle world, whose effectiveness in different environment settings is evaluated to demonstrate the effectiveness of the agent and its feasibility in terms of aiding student learning of RL concepts. Server-side metrics are also computed to verify the effectiveness of the system in handling communication. To use this system as part of game-based learning in an AI course at the undergraduate level, parts of the agent will be given to students, and the rest of the code will be their task to write. As they build parts of the system they will get feedback via backend logs to see whether their agent is learning to navigate the puddle world environment effectively. The system as a whole is evaluated for its feasibility based on the training termination conditions for the RL agent to find the ideal settings in supporting student learning of RL concepts involved. This thesis

also briefly touches on the cross-language integration of Python code dealing with the main AI algorithms for an agent into a Java-based client-server application that plays the Game of Amazons to make it easier for students to code in with the use of Python, which they may be more comfortable with, and will enable them to use machine learning libraries defined in Python in their code.

Of the remaining chapters of this thesis, chapter 2 will focus on describing concepts of AI, notations for RL, the algorithms in RL, literature reviews of the existing platforms for RL agent development, and existing frameworks for integrating Python code with Java code. Chapter 3 will begin by describing the structure and aspects of the puddle world in PNRL, the architecture of PNRL, and will end with a brief description of the architecture of the Python-Java integration for the Game of Amazons. Chapter 4 will put the server-side and client-side RL agent's effectiveness under different environment conditions and training termination conditions to the test. Chapter 4 will also go over the results of these experiments while interpreting the results in the context of supporting student learning of RL concepts. Chapter 5 will conclude the thesis by mentioning the work done in the thesis and make recommendations for potential future work on the research conducted.

Chapter 2

Background and Preliminaries

Artificial Intelligence has been defined as giving computers the ability to think, reason, and use their learning to complete tasks [13]. 75 to 375 million people will have their jobs replaced by Artificial Intelligence by 2030 [14], demonstrating the rising adoption of AI as part of society and modern workflows. The goal of developing"narrow AI" [15, 16] systems that can learn from domain-specific data and conduct actions is to delegate these algorithmic tasks, and focus human intelligence on automation of tasks that require more than just algorithmic thinking [17]. Focusing on building systems that can reason efficiently in a variety of domains would mean focusing on constructing an Artificial General Intelligence (AGI) system. General intelligence has no unifying definition but consists of: being able to perform tasks in different situations, suggest solutions to problems that the inventor may not have thought of, and possess the ability to learn concepts from one situation and apply it to a multitude of different situations [15].

Current research in AI is highly focused on different areas of narrow AI, with Machine Learning (ML), and Reinforcement Learning being at the forefront. Machine Learning is the ability for machines to learn insights from data specific to a domain using labeled or unlabeled data. Reinforcement learning consists of the AI system, or the agent, performing sequential tasks and making many sequential choices in order to achieve an end goal, while ensuring that a function known as a reward function is maximized, in turn leading to minimal negative outcomes. Algorithms in reinforcement learning focus on learning strategies to solve problems in their environment, known as policies [18], and weighing trade-offs between whether the agent should continue exploiting actions that they know the outcome of versus exploring new actions that they do not know the outcome of. This is commonly known as the exploration-exploitation trade-off [18]. It deals with how much of the agent's focus is on exploiting actions whose outcomes aim to increase immediate rewards, and how much of the agent's focus is on exploring new actions whose outcomes are unknown aiming to increase future rewards in the environment. Algorithms that learn how to balance the exploration-exploitation trade-off tend to succeed in completing their goal. The training process for RL agents focuses on learning optimal policies which can then be used to test the effectiveness of the agent in a test environment. This helps in checking whether the RL agent's learning can generalize to different environments.

2.1 Introduction to Reinforcement Learning

2.1.1 Markov Decision Processes

To explain how decisions affect future rewards, and states the agent could potentially enter, one requires a structure capable of holding this information for an RL agent in an environment. This is what a Markov Decision Process (MDP) functions as. An MDP is defined over a state space, and at any given time, an MDP is in one of the states. Associated with each state is a set of actions that the agent can perform. For each action performed from a state, the agent receives a reward. MDPs are central to understanding reinforcement learning and aim to learn a model of the MDP and a strategy to navigate an environment [1]. The state space S is the state space that an agent can possibly traverse through during its traversal of the environment. S^+ represents all the states including the terminal state [1]. The action space A is the set of actions an agent can perform from any state in the state space. The reward space R is the set of rewards an agent can obtain from making state transitions in its traversal of the environment.

In this thesis, the focus will be on a finite state-action-based puddle world which can be modeled as a discrete time MDP in a discrete state space. Given the initial state at t=0, the agent performs a sequence of actions, resulting in a sequence of rewards and successive states which can be defined as such [1]:

$$\{S_t, A_t, R_t;\ t \geq 0\} \tag{2.1}$$

It is assumed that this sequence of Random Variables (RVs) satisfies the Markov property; i.e. the probability distributions of the current state and reward are dependent entirely on the previous states and actions [1]. The probability of transitioning to state s' with reward r from a previous state s after performing action a is defined as the following conditional probability [1]:

$$p(s', r|s, a) = p(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{2.2}$$

The expected rewards for any state-action-next-state triplets between time steps t and $t-1$ as a weighted summation over all possible rewards is [1]:

$$r(s, a, s') = E[R_t\,|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r\ *\ \frac{p(s', r|s, a)}{\sum_{r \in \mathcal{R}} p(s', r|s, a)} \tag{2.3}$$

A central assumption to RL is that the agent knows some information about its environment [1], and more knowledge on the environment is obtained through the agent's experience of reacting to its surrounding states' rewards to maximize the expected value of the final cumulative reward obtained. The reward r assigned to states that are favorable are positive in value, and rewards assigned to states that are not favorable are negative in value. This allows the agent to learn specific strategies to reach a goal state s_{goal} based on the immediate

feedback provided by transitioning to state s' and the reward r of that state. The expected final cumulative reward summed from time steps t (initial state) to T (final state) can be defined as follows [1]:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{2.4}$$

This formulation of the cumulative reward fits perfectly into applications where a sequence of steps leads to a final state known as the terminal state. This sequence of steps where the agent interacts with the environment is called an episode [1]. The terminal state can either be the goal state or simply the state the episode was terminated in. Each episode is independent of the next episode, similar to if one were to play a game till they lost or won, and then played the game again [1]. Tasks that can be split into episodes are known as episodic tasks, and forms the basis of the puddle world explained in the next chapter. In contrast to this, there exist tasks that have no terminal state, such as creating an agent with a goal to live life like a human does and attempting to maximize the cumulative reward of the agent, which are known as continuing tasks [1].

Evaluating rewards from both long-term and short-term perspectives is important to ensure an agent's effectiveness in maximizing cumulative reward while attempting to reaching the goal state, and this is where the discount factor γ is vital [1]. The discount factor is a value between and including 0 and 1, which handles this by decreasing the weight of rewards attained k time steps from the current time step by a factor of γ^{k-1} [1]. If one were to assume a γ value close to 0, the agent would give a lot more importance towards seeking immediate rewards, and very little importance towards seeking future rewards from states that may not have been explored yet. And conversely, if one were to assume a γ value close to 1, the agent would give a lot more importance towards seeking future rewards, and very little importance towards seeking immediate rewards.

Integrating the discount factor into the final cumulative reward equation (equation 2.4), the following formula can be deduced (which from now will be known as the final discounted cumulative reward equation and the function that the agent attempts to maximize) for G_t, where the discount factor is multiplied $k-1$ times to the reward at time step k and summed with the rewards from previous time steps [1]:

$$G_t = \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

The value of G_t has a specific pattern that can be noticed from equation 2.5, and is substituted to simplify the expression further [1]:

$$G_t = R_{t+1} + \gamma(R_{t+2} + \gamma^2 R_{t+3} + ...) \quad = R_{t+1} + \gamma * G_{t+1} \tag{2.6}$$

The final discounted cumulative reward equation can be unified to cover episodic and continuing tasks by replacing the regular terminal state with an absorbing state [1], which continues to add a reward of 0 forever in the case of an episodic task after having reached the terminal state. And in the case of a continuing task, the agent will never reach this state.

The modified representation is as follows considering a summation from time steps $t+1$ till step T:

$$G_t = \sum_{k=t+1}^{\infty} \gamma^{k-t-1} R_k \tag{2.7}$$

RL algorithms require computing values of a function that measures the efficacy of performing action a in state s to transition to state s' in terms of expected future discounted cumulative rewards, and these functions are known as value functions [1]. An agent's policy, represented as π, is defined as pairings of probabilities of choosing a particular action to perform from a given state, and forms the general strategy that the agent uses to navigate its environment. The value function of state s under a policy π, represented by $v_\pi(s)$ is the expected final discounted cumulative reward for an agent that starts in state s and follows the policy π from thereon [1]. The formulation of the value function is as follows [1]:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \,|\, S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|\, S_t = s] \qquad \forall s \in S \tag{2.8}$$

The value of the action-value function represents the expected discounted cumulative reward from following policy π and selecting action a from state s [1]. The formulation of the action-value function is as follows [1]:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \,|\, S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|\, S_t = s, A_t = a] \tag{2.9}$$

These value and action-value functions are estimated from the sequence of interactions that an agent has with its environment by averaging the value function's magnitude for all the successive states from state s or $v_\pi(s)$. By tracking separate averages for each action taken from each state, these values will converge to the action-values from $q_\pi(s,a)$. This averaging method is known as monte carlo (MC) estimation, and is practical in the case where there are a limited number of states, such as in a puddle world [1]. Monte Carlo estimation enables the agent to learn directly from experience of interacting with environment without an understanding of the dynamics of the environment.

An important property of value functions is that they can be defined recursively in a similar way to the cumulative discounted reward function G_t [1]. The deduction of this recursive relationship comes from substituting equation 2.6 into the expected value calculation, considering the policy that an agent uses to pick the next state it transitions to, multiplying it by the probability of transitioning from state s to s' based on the reward r, and summing it with the successive states' rewards. The value of x from the expected value ($E[x]$) formula corresponds to the $r + \gamma v_\pi(s')$, and $P(X = x)$ corresponds to $\pi(a|s) * p(s', r|s, a)$ in the derived formula below. The deduction is as follows [1]:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t \,|\, S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma\, G_{t+1} \,|\, S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1} \,|\, S_{t+1} = s']] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)[r + \gamma v_\pi(s')] \qquad \forall s \in S
\end{aligned} \tag{2.10}
$$

The final line in the formula derived above is called the Bellman equation, and represents the relationship between the values of a state s and all of its possible upcoming states s' [1]. The essence of the Bellman equation is that it is the value of the starting state s following a policy π is equal to the discounted next state value summed with the reward gained by transitioning to that next state $(r + \gamma v_\pi(s'))$. This sum is then averaged over all possible values for the upcoming states multiplied by the probabilities of those values occurring $(\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a))$. The value function (v_π) is the unique solution to the Bellman equation for state s.

A policy π can be deemed as being better than another policy π' if its expected final discounted cumulative reward $v_\pi(s)$ is greater than or equal to the other policy's expected final discounted cumulative reward $v_{\pi'}(s)$. The policy with the highest expected discounted cumulative reward possible is known is the optimal policy, and is represented as π_*. The optimal state-value function is represented by v_* and is formulated as follows [1]:

$$
v_*(s) = \max_\pi v_\pi(s) \qquad \forall s \in S \tag{2.11}
$$

The optimal action-value function is represented by q_* and is formulated as follows [1]:

$$
\begin{aligned}
q_*(s, a) &= \max_\pi q_\pi(s, a) \qquad \forall s \in S \;\; and \;\; \forall a \in A(s) \\
&= \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \,|\, S_t = s, A_t = a]
\end{aligned} \tag{2.12}
$$

The Bellman optimality can be reformulated as follows to show that a value of a state under an optimal policy is the highest expected return for the best action from that state [1]:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_\pi(s, a) \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1}) \,|\, S_t = s, A_t = a] \\
&= \max_a \sum_{s', r} p(s', r|s, a)[r + \gamma v_*(s')]
\end{aligned} \tag{2.13}
$$

The Bellman optimality equation for the optimal action-value function is as follows [1]:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\
&= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')]
\end{aligned} \tag{2.14}
$$

Solving the Bellman optimality equation is seldom feasible as it requires a lot of computation for all states' occurrence probabilities, requiring full understanding of the dynamics of the environment, and an assumption that the Markov property is satisfied [1]. Therefore, approximate solutions are most efficient in terms of computing power and memory available.

2.1.2 Dynamic Programming

Dynamic Programming is a set of algorithms that accurately compute optimal policies under the assumption that a perfect model of the environment's dynamics are provided in the form of an MDP [1]. Other algorithms attempt to perform close to DP with lesser computational cost by dropping the assumption of a perfect environment. Equations 2.13 and 2.14 can be used in deducing update rules in DP algorithms. The calculation of the value function v_π under policy π is known as policy evaluation or the prediction problem [1], and the deduction can be seen in equation 2.10. Assuming the environment's behavior is fully known, the computation of solutions to the value function for each state is compute-heavy but achievable, and iterative methods to approximate value functions for a state s are the best approach [1]. The value function for each state is iteratively updated based on the discounted estimates of the value function for the previous iteration, summed with the current reward, and multiplied by the probability of each of those values occurring using the Bellman equation for the value function as an update rule. This can be formulated as follows: [1]:

$$
v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \qquad \forall s \in S \tag{2.15}
$$

This update rule, known as an expected update, has been proven to cause v_k to converge to v_π as k approaches infinity as long as $\gamma < 1$ or agent traversal termination happens from any state assuming policy π [1]. The complete algorithm for iterative policy evaluation is shown in Algorithm 4 in Appendix B [1].

Calculating the value function under a policy π is useful in determining policies that increase final discounted cumulative reward. To determine policies this reward, the policy improvement theorem must be considered, and its general formulation is as follows [1]:

$$
q_\pi(s, \pi'(s)) \geq v_\pi(s) \quad \Rightarrow \quad v_{\pi'}(s) \geq v_\pi(s) \tag{2.16}
$$

The policy improvement theorem states that if it is better to select an action a from state s based on policy π' in terms of maximizing final discounted cumulative reward, than it is to follow a policy π from state s to the terminal state, then policy π' in the worst case is just as good as policy π or better than π [1]. The formulation of the policy that leads to policy

improvement is the action that maximizes the expected discounted cumulative reward, and can be represented as [1]:

$$\pi'(s) = \underset{a}{argmax} \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.17}$$

[1]

Using v_π, an improved policy π' can be derived. π' can be used to compute and evaluate $v_{\pi'}$, which can then be used to derive an even more improved policy π'', until an optimal policy is reached in a finite number of iterations. This is known as policy iteration [1]. The algorithm for policy iteration is shown in Algorithm 5 in Appendix B [1].

The policy evaluation step in policy iteration can be stopped after a single update of each state's value function computed values instead of waiting for convergence to v_π. This shorter version of policy evaluation along with policy improvement is known as value iteration [1]. Value iteration essentially merges the value computation step in policy evaluation and the policy computation step in policy improvement into one update rule summed over all actions that can be used to determine the action yielding the highest expected discounted cumulative reward. This update step can be formulated as follows [1]:

$$v_{k+1}(s) = \underset{a}{max} \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{2.18}$$

The corresponding value iteration algorithm is shown in Algorithm 6 in Appendix B [1].

Dynamic Programming is not feasible for problems with large state spaces, but compared to other methods for solving MDPs, they are computationally efficient. The worst case time complexity of finding an optimal policy with DP methods is polynomial time with respect to n possible states and k possible actions even considering the large total number of possible deterministic policies is k^n [1]. The computational complexity of DP methods increases exponentially as the number of state variables holding information about the state increase, but DP methods still beat linear programming and direct search methods in computational efficiency in large state space problems [1].

2.1.3 Temporal Difference Learning

Temporal Difference (TD) learning merges ideas from Monte Carlo methods with ideas from DP methods. TD methods are similar to Monte Carlo methods in that they learn from experience in the environment without a model of the environment's dynamics [1]. And, they are similar to DP methods in that they make updates to their approximations of values based on the other learned approximations [1]. TD methods' approach to the prediction problem is what differentiates these methods from DP and Monte Carlo methods, as all of these methods implement a variant of a Generalized Policy Iteration (GPI) to solve the control problem of finding an optimal policy [1]. A TD method known as $TD(0)$ or one-step TD method waits only till the end of the next time step $t + 1$ to make an update to its estimate of the value function for a state S_t using the upcoming reward R_{t+1} to estimate $V(S_{t+1})$, and can be formulated as follows [1]:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \tag{2.19}$$

This approach differs from MC methods in that MC methods wait until the end of an episode before making an update to their value function for a state S_t [1]. The $TD(0)$ method's algorithm is as follows [1]:

Algorithm 1 TD(0) - estimation of the value function V via v_π

Input: Policy π for evaluation
Parameter: Step size (how much Q-values are updated by) [27, 28] $\alpha \in (0, 1]$
Initialize $V(s)$ with arbitrary values $\forall s \in \{\mathbb{S}^+ - s_{terminal}\}$ and $V(s_{terminal}) = 0$

Loop for each episode:
    Initialize S
    Loop for each step in the episode:
        $A \leftarrow$ action chosen by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha \left[R + \gamma V(S') - V(S) \right]$
        $S \leftarrow S'$
Until S is terminal

Due to the fact that $TD(0)$ methods make updates to their estimates based on existing estimates, they are called bootstrapping methods [1]. This is similar to how DP makes updates to its estimates. The formulation of the estimated v_π as a recursive form of the Bellman equation is as follows [1]:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \,|\, S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma\, G_{t+1} \,|\, S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \,|\, S_t = s] \end{aligned} \tag{2.20}$$

MC methods make use of the value attained from the first line of equation 2.20 as a goal reward for one step ahead and the discounted future reward using sampled final discounted cumulative rewards. DP methods on the other hand use the current estimates in the form of the last line of equation 2.20 as a target reward [1]. The target return for TD methods uses the estimate of the value function V in the current time step, along with the sampled expected values to estimate v_π [1]. To measure how effective a particular TD method is in determining the final reward from being in a state S_t, the TD error, δ_t, is used [1]:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2.21}$$

TD methods not requiring a model of the environment's probability distributions of rewards and transition probabilities makes them useful in applications where DP methods are not. TD methods' estimates being incremented by the next time step instead of at the end of an episode like MC methods, makes them highly useful in applications that have long episodes. TD(0) under a constant policy π has been proven to converge quicker to v_π than

MC methods holding the step size constant for tasks that are stochastic in nature [1]. TD methods find estimates of the value function (the parameter in question) that match the maximum-likelihood model (estimates parameters that maximize the probability of observing the training data) of the MDP at every time step, while MC methods compute estimates to reduce the mean square error with respect to the training data at the end of each training episode. This is the reason behind why batch-training-based TD methods converge quicker than batch-training-based MC methods.

There are two types of control problem solutions to finding an optimal policy, on-policy methods and off-policy methods [1]. On-policy methods use the same policy that explores the environment for policy improvement while attempting to find an optimal policy [1]. Off-policy methods contain a policy that is used to explore the environment, and a different policy used for policy improvement [1]. On-policy TD control methods consider transitions from one state-action pair to the next and aim to learn an action-value function (Q) instead of a value function (V). One such formulation of this is known as Sarsa, as it uses all of $S_t, A_t, R_t, S_{t+1}, A_{t+1}$ within its update rule formula as follows [1]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) \; + \; \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right] \tag{2.22}$$

The Sarsa algorithm (dependent on the policy's behavior) using an ϵ-greedy policy chooses an action with the highest action-value function magnitude or a random action with a probability of ϵ. The algorithm is as follows [1]:

Algorithm 2 Sarsa (On-policy TD control method) - estimation of the action-value function Q via q_*

Parameters: Step size $\alpha \in (0, 1]$ and a small $\epsilon > 0$ for epsilon-greedy action selection with epsilon decay to encourage more exploitation as training progresses

Initialize $Q(s, a) \;\; \forall s \in \{\mathbb{S}^+ - s_{terminal}\}$ and $a \in \mathcal{A}(s)$ with arbitrary values and $Q(s_{terminal}) = 0$

Loop for each episode:
- Initialize S
- Choose an action A from S using a policy such as ϵ-greedy policy deduced from Q
- Loop for each step in the episode:
  - Take action A, observe R, S'
  - Choose an action A' from S' using a policy (ϵ-greedy policy) deduced from Q
  - $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A) \right]$
  - $S \leftarrow S'$
  - $A \leftarrow A'$

Until S is terminal

An off-policy TD control algorithm, known as Q-learning estimates the optimal action-value function without depending as much on the policy, as it selects the action with the highest action-value function magnitude without regard to the policy's behavior from the states that the policy chooses to explore. The Q-learning update rule is as follows [1]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) \; + \; \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.23}$$

The Q-learning algorithm is as follows [1]:

Algorithm 3 Q-learning (Off-policy TD control method) - estimation of the action-value function Q via q_* and using the estimation to determine π_*

Parameters: Step size $\alpha \in (0, 1]$ and a small $\epsilon > 0$ for epsilon-greedy action selection with epsilon decay

Initialize $Q(s, a) \;\; \forall s \in \{\mathbb{S}^+ - s_{terminal}\}$ and $a \in \mathcal{A}(s)$ with arbitrary values and $Q(s_{terminal}) = 0$

Loop for each step in the episode:

Initialize S

Loop for each step in the episode:

Choose an action A from S using a policy (ϵ-greedy policy) deduced from Q

$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right]$

$S \leftarrow S'$

Until S is terminal

This section has now provided the fundamental knowledge required to understand the basis of the system introduced in the next chapter, and the various reinforcement learning platforms mentioned in the next section.

2.2 Existing platforms for Reinforcement Learning environments

Before describing the specifics of PNRL in the next chapter, it is necessary to review the landscape of other existing platforms for reinforcement learning environments to understand the differences between the various existing platforms, their advantages, and why PNRL is more appropriate for the use case of supporting student learning of RL concepts in an undergraduate level AI course.

A platform for RL agent evaluation is necessary to evaluate the effectiveness of an agent, and to deduce whether the agent can be applied in different environments. Benchmarks for evaluating RL agents are used to evaluate its learning performance in an environment. The performance of different agents can be compared on a level playing field with the benchmarks to improve State of The Art (SoTA) RL agents. In this literature review, the focus is on three RL agent evaluation platforms that can be used for agent performance evaluation, with some of them coming with built-in benchmarks.

ArtBot, is a simple supervised and reinforcement learning platform for teaching concepts in ML to primary and secondary education students. Its main purpose is to integrate student experimentation of ML & RL concepts by seeing them in action, and digital gamification of the concepts to aid student learning [5]. The main advantage of a system like this is

to get younger students into the habit of thinking algorithmically and to ponder about the design of AI systems in the real-world, hence, not going too deep into the specifics of the implementations of the algorithms required to physically build these systems that would be expected of an undergraduate student.

Arcade Learning Environment (ALE) is a platform for developing and validating RL models' efficacy in terms of their ability to plan tasks, learn based on the environment via RL, and transfer learning [9]. This platform contains Atari 2600 emulator-based games that are converted into a reinforcement learning problem by maintaining a score for the game and whether the game has terminated or not. This provides benchmark results to compare custom agents' performance against one another [9]. The bench marked results were recorded using a SARSA(λ) TD control method, which is an extension of Algorithm 2, the SARSA algorithm, from section 2.1.3. This algorithm updates not just the current state-action pair's state-action value estimate, but a fraction of the total past state-action pairs' state-action value estimates, based on the λ factor [9]. This comes in handy in situations where the effect of actions is delayed by a few or many time steps. The results from running agents trained on a group of training games and then tested on a group of testing games indicate that ALE is a good platform for complex game-based RL algorithm applications that involve using training RL models and then testing them in a similar setting [9]. The main advantage of a platform such as ALE is the amount of complexity that can be accounted for by the games that agents are trained on (in comparison to a puddle world-based game), which can be expected to be quite advanced and may not be suitable for an undergraduate curriculum in reinforcement learning.

Project Malmo is a testbed for AI agents in a continuous Minecraft world for supporting development of agents or algorithms in computer vision, robotics, planning of tasks, and multi-agent systems [10]. Malmo's environment is dynamic, has many agents interacting in it, and multiple-step tasks where planning can be required considering that the agent's computational resources are limited [10, 19, 20]. Such a testbed can be used to test even more complex RL algorithms than the ones that could be tested with ALE, which can be expected to be quite advanced and may not be suitable for an undergraduate curriculum in reinforcement learning.

From the three platforms described, ArtBot's simple agent setup and game play for secondary school students is lacking in amount of detail to aid learning of reinforcement learning concepts at the undergraduate level. ALE and Project Malmo's complex game details and complex multi-agent interactions respectively provide excessive amounts of detail to aid learning of reinforcement learning concepts at the undergraduate level. In the next chapter, a system which provides the right amount of detail for undergraduate students to learn and implement reinforcement learning algorithms is proposed, called PNRL. PNRL's purpose will be to enable students to build parts of an RL agent that learns to navigate a puddle world. PNRL will be specifically focused on teaching students how to implement the Q-learning algorithm in an RL agent within a discrete environment, and not on complex multi-agent interactions in continuous environments or simply introducing RL concepts at a surface level. Therefore, PNRL will provide the perfect middle ground in terms of complexity of RL concepts for undergraduate students to learn with the hands-on approach of building

the algorithmic parts of the RL agent. PNRL was built using Java [21], SmartfoxServer server API [22] and Client API in Java [23], with Maven [24] as the dependency and build manager.

2.3 Python-Java cross-language integration platforms

Before describing the specifics of the Python-Java code integration to play the Game of Amazons in the next chapter, it is necessary to review the most relevant platforms that can be used to develop a Java to Python cross-language integration by understanding the differences between the various existing platforms, their advantages, and why Py4J is more appropriate in this context of the UBC COSC 322 AI course.

The UBC COSC 322 course's project has a SmartFoxServer server API [22] that allows students to communicate a move in the Game of Amazons (that contains the following: current queen position, new queen position, and arrow position in the grid) from their client to the SmartFoxServer client API [23]. The client API then communicates the moves to the server. The other player will be able to receive the move and make their own in the same way. The server and client-side code are written in Java, and the SmartFoxServer client API does not have Python support. This is a disadvantage for students, as they cannot apply machine learning libraries to perform complex computations that are easily available through Python libraries. Instead, they need to write the algorithms from scratch in Java, which can be tricky. The availability of ML libraries on Python and lack of a Python client API support creates the need to build a Python-Java integration for the Game of Amazons project.

Java Embedded Python (JEP) is a platform that has Python embedded into Java via the Java Native Interface (JNI) and Python API, allowing for Python code to be developed alongside Java code. The code is accessible via Java through the API of JEP [25] while spinning up a Python interpreter in the Java Virtual Machine (JVM). This approach has the advantage of efficient data transmission between Java and Python since they share the same memory space, and JEP's API can be used to access the computations for the Game of Amazons done by Python through Java to communicate the next move to the Java SmartFoxServer server API for further processing.

Py4J is a platform that allows for Python code to access Java objects dynamically through a JVM gateway [26]. The advantage of this method is that it is easier to setup for students as they do not need to worry about interpreters, and does not add on much overhead in terms of sending computations to Java. It also provides students the ability to make their Python code as scalable as they need without affecting the Java code's performance.

From the two platforms described above, Py4J is easier for students to setup as they can use a simple Java gateway from Python. This will allow students to write code in Python for the computations using the vast AI algorithm libraries, and send their computations to the Java client-side code, which communicates them to the Java SmartFoxServer server API for UBC's COSC 322 course.

Chapter 3

PNRL

3.1 Puddle World

The puddle world within PNRL is the discrete grid-world environment in which the RL agent constructed on the client-side will traverse through starting from a state ID of 0 (top-left of the grid world), and aiming to reach the goal state located at the state ID: $gridSize^2 - 1$ (bottom-right of the grid world). The server-side will hold a representation of this world as well. The agent within PNRL can perform the following actions of moving in $A = \{$up, down, left, right$\}$ directions to reach the goal state, but cannot move diagonally. The RL agent (depicted below inside the blue state) has an objective is two-fold in that it has a soft goal of avoiding the puddles, and a hard goal of maximizing the discounted cumulative reward in attempting to reach the goal state.

The PNRL puddle world has been designed to mimic the puddle world studied by Richard Sutton [29]. The puddle world studied by Sutton was designed for a continuous state space, with the agent being initialized in a random state and the puddle states initialized together in chains[29]. Figure 3.1 depicts the puddle world that Sutton studied [29]. PNRL was built to serve as a discrete state space version of the puddle world studied by Sutton, while preserving the main ideas of the puddle world such as the puddle regions, goal state, and rewards received at each step. PNRL's puddle world has been designed to be intuitive (row-major 2-dimensional array configuration) for students at the undergraduate level. It initializes the puddles in groups at random such that there is a consistent shape to the sets of puddle states compared to the puddle world Sutton was studying that had two puddle regions of various shapes [11]. The puddle states in PNRL are grouped in squares of size $w \times w$, where w is the size of the contiguous blocks of width or height that forms a grouped square of puddle states (i.e. puddle size).

For example, let us assume a puddle world with a default reward of -0.02 for transitioning to any non-terminal state, a reward of -2.0 for transitioning into a puddle state, and a reward of $+20.0$ for transitioning into a goal (terminal) state with a grid size (number of total states in the puddle world, or puddle-world size) of $5x5$, and two 2×2 blocks that serve as the groups of puddle states. To represent the PNRL puddle world visually, let us assume that the initial state for the RL agent is colored blue, the non-terminal states are colored white,

the goal state is colored green, and the states colored red are the puddle states that the RL agent is attempting to learn how to avoid. Figure 3.2 depicts this example configuration of the PNRL puddle world.

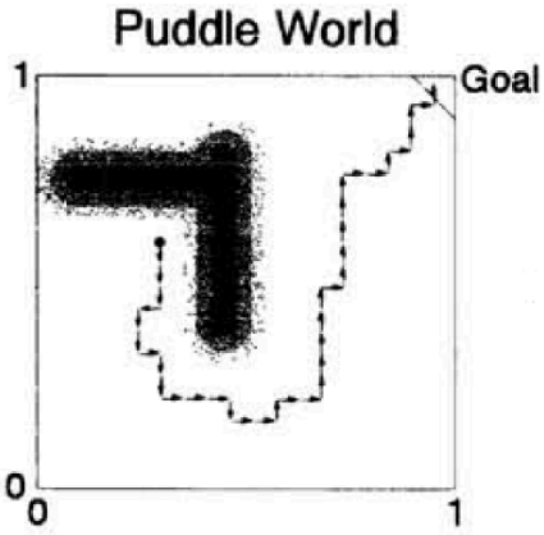

Figure 3.1: Puddle world studied by Sutton [29]

	-0.02	-2.0	-2.0	-0.02
-0.02	-0.02	-2.0	-2.0	-0.02
-0.02	-2.0	-2.0	-2.0	-0.02
-0.02	-2.0	-2.0	-0.02	-0.02
-0.02	-0.02	-0.02	-0.02	+20.0

Figure 3.2: PNRL puddle world

There are several factors that can affect the RL agent's learning process, success in reaching the goal state, and maximizing the discounted cumulative reward. They are described in table 3.1.

Table 3.1: Factors affecting RL agent learning process and success

Factor	**Brief description**
Puddle-world size	See definition above
Number of puddles & individual puddle size	See definition above
Number of training episodes	See section 2.1.1
α, γ, and ϵ	Step size, Discount factor, and Probability of picking a random action
Default reward	Reward received by the agent on entering a non-puddle state or terminal state
Puddle state reward	Reward received by the agent on entering a puddle state
Goal state reward	Reward received by the agent on entering the terminal/goal state
Training termination condition	An integer from 0 to 2 that picks the termination condition of an agent's training within an episode
Maximum number of steps in a training episode	Max number of state transitions an agent can make within one training episode
Training termination probability	Probability with which the agent's training is terminated immediately
Success reward threshold	Parameter set based on preference to check if the agent has succeeded in its objective

The puddle world's size plays a big role in whether the agent reaches the goal state in a reasonable time frame due to a larger grid requiring many more "correct" decisions to be made by the agent to reach the goal state. The number of puddles and the individual puddle size are also a major factor in the agent's learning process as reward feedback from stepping into puddle states is a direct part of the agent's learning process. The number of training episodes controls the amount of experience an agent acquires in interacting with the puddle world. An increase in the number of training episodes drives an increase in ability of the agent to avoid puddles and reach the goal state while maximizing discounted cumulative reward.

The parameters α, γ, and ϵ directly influence the agent's learning process by determining how much of the Q-value is updated for an action-state pair, the amount of preference given to future rewards in comparison to immediate rewards, and the probability that the agent selects a random action (exploration) in comparison to picking the action with the highest Q-value (exploitation) respectively. The parameter ϵ offers a balance in the exploration-exploitation tradeoff, which is part of many successful RL algorithms. The ϵ value is decayed over every step the agent takes enabling the agent to raise its confidence in its action-value function's decisions, reducing the number of suboptimal decisions made with an increasingly smaller probability of exploration, and sidestepping the problem of over-exploration [30]. The default, puddle and goal state rewards all play a part in dictating how strong a signal is provided to the agent at every step. More extreme values for the rewards generally cause

the agent to learn how to avoid puddles faster. The training termination condition within PNRL also plays a part in the agent's success rate, as PNRL has three training termination conditions (detailed in section 3.3), which choose when to terminate an agent's learning process, therefore, directly influencing the amount of training episodes and steps the agent learns for. The maximum number of steps in a training episode is a parameter that is a part of one of the training termination conditions in PNRL and directly influences how long the agent will train for within an episode. The training termination probability is also a parameter that is a part of one of the training termination conditions in PNRL and directly influences how long the agent will train for within an episode. The success reward threshold does not affect the agent's learning process, but can serve as a useful method of assessing whether the agent is maximizing the cumulative discounted reward in attempting to reach the goal state within an episode.

3.2 PNRL Architecture

PNRL's architecture is client-server-based with the server-side built using the Java SmartFoxServer server API, and the client-side is built using the Java SmartFoxServer client API. The architecture of PNRL was built in this manner to maintain a consistent architecture pattern as that of the Game of Amazons project in the COSC 322 course at UBC for ease of integration into the COSC 322 API. The client-server architecture delegates the action selection and agent training logic to the client-side for students to complete. The maintenance of a master puddle world and q-table, validation of the agent's actions, and monitoring of the training process of the agent was delegated to the server-side. The monitoring of the client-side agent's learning process from the server-side was made possible through defining a clear messaging protocol enabling communication transparency of data. The communicated data consists of: rewards, states, actions, training completion, and q-table (which states as rows, actions as columns, and all values initialized to 0) updates made after every step taken by the RL agent in an episode. This transparency also makes it easier for students to see Q-values for state-actions pairs stabilizing over training episodes with the epsilon-greedy policy. The students can see the progress of the RL agent's training and learning process through client-side logs of each step taken by the RL agent. Server-side logs are also kept holding a copy of the client-side agent's learning updates and additional logs of responses sent back to the client. The server-side logs produced appear on the server-side instance of SmartFoxServer Java server terminal logs after running the program from the client-side, allowing instructors to view the learning process of the agent a student builds parts of, while students can view the client-side logs from the terminal they run their agent from. The client-side logs mirror the server-side's logs and contain the Q-value updates, cumulative discounted rewards, and data specific to the agent's state transitions to view and improve their agent's learning process.

3.2.1 Communication Protocol

The communication protocol in PNRL was defined to encapsulate the following RL-specific variables important in ensuring the agent's learning process is conducted in accordance with

the Q-learning formula defined in equation 2.23 and below [1]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) \; + \; \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{3.1}$$

The communication protocol is bidirectional. A breakdown of the client-to-server and server-to-client messages with a description of the data communicated in each message is as follows:

Table 3.2: Client-to-server messages

Message	**Description**
1. GAME_STATE	This message contains a request for the state ID of the current state of the agent.
2. GAME_AVAILABLE_ACTIONS	This message contains a request for the list of available actions for an RL agent from the current state.
3. GAME_AVAILABLE_REWARDS	This message contains a request for the list of available rewards with the respective available actions for an RL agent from the current state.
4. GAME_ACTION_MOVE	This message contains a request for the action (up, down, right, left) performed by the agent and the state ID of the current state of the agent.
5. GAME_ACTION_REWARD	This message contains a request for the action (up, down, right, left) performed by the agent, the reward attained from the action taken by the agent, and the state ID of the successive state of the agent after having performed the action.
6. GAME_FINAL_STATE	This message contains a request for a boolean value that notifies the server whether the client-side agent has reached the terminal state, the discounted cumulative reward and the number of steps taken by the agent in the current training episode.
7. GAME_RESET	This message contains a request for the server to reset the RL environment (setting discounted cumulative reward to 0, and moving the agent back to state 0) after an episode of training with the user's name.

Continued on next page

Message	Description
8. GAME_Q_UPDATE	This message contains a request for the list of state IDs for which the Q-values are updated, a list of action indices for which the Q-values are updated, and the corresponding updated Q-values for the state-action pairs.
9. GAME_INFO	This message contains a request for a summary of the recently concluded training episode with the discounted cumulative reward, steps taken by the agent in the episode, total number of training episodes uptil now, and the number of successful episodes.
10. GAME_TRAINING_COMPLETE	This message contains a request for the server to verify that the client-side agent has finished training.

Table 3.3: Server-to-client messages

Message	Description
GAME_STATE_RESPONSE	This message is a response to the client's request containing the state ID of the current state of the server-side representation of the agent.
GAME_AVAILABLE_ACTIONS_RESPONSE	This message is a response to the client's request containing the list of available actions for the client-side agent from the current state.
GAME_AVAILABLE_REWARDS_RESPONSE	This message is a response to the client's request containing the list of available rewards to the client-side agent for the corresponding available actions from the current state.
GAME_ACTION_REWARD_RESPONSE	This message is a response to the client's request containing the action taken by the server-side representation of the agent, the reward received by performing the action, and state ID of the successive state after having performed the action.

Continued on next page

Message	Description
GAME_FINAL_STATE_RESPONSE	This message is a response to the client's request containing a boolean value that notifies the client whether the server-side representation of the agent has reached the terminal state, the discounted cumulative reward and the number of steps taken by the server-side representation of the agent in the current training episode, total number of training episodes uptil now, and the number of successful episodes.
GAME_INFO_RESPONSE	This message is a response to the client's request containing the discounted cumulative reward and the number of steps taken by the server-side representation of the agent in the current training episode, total number of training episodes uptil now, and the number of successful episodes.
GAME_ERROR	This message is a notification sent to the client when there are missing fields in a request and when there are inconsistencies within the client and server's learning process.

The messaging protocol defined above was engineered to ensure only essential data for the agent's training was communicated from client to server. This enables the performance of the system to scale efficiently with larger training episode counts, and minimizes the number of communication errors while allowing for efficient Q-value updates and agent training. This messaging protocol along with the environment variables (defined in 3.1) provide the client-side RL agent with everything required to implement the Q-learning algorithm, which is defined in the previous chapter in Algorithm 3.

3.2.2 Summary of Client & Server Functions

PNRL's client-side functions to support action selection through Q-learning with an epsilon-greedy policy, by selecting either the action with the maximum Q-value or a random action in accordance with Algorithm 3 until the training termination condition defined for an episode is hit. The client-side communicates the messages defined in the client-to-server messaging protocol and processes the server's responses. It also internally maintains a representation of the puddle world, and moves its representation of the agent within the puddle world. A persistent q-table of the Q-values of all state-action pairs that the agent traversed over x episodes within a single run of the program is maintained. This q-table is updated after each action the agent takes.

PNRL's server-side functions to maintain and update a representation of the client-side agent's movements and learning process within a master puddle world and q-table maintained

here. The server-side responds to all client requests with the messages defined in the server-to-client messaging protocol. The server-side validates the client-side agent's actions made by ensuring that the agent stays within the bounds of the puddle world and the rewards received from each state is accurate. The server-side receives q-table updates from the client-side and stores the updated values for learning process monitoring purposes. The cumulative discounted rewards per episode, the number of episodes, and the number of successful episodes are all monitored and updated after receiving moves from the client-side agent.

This clear separation of concerns between the client and server-side roles allows for students to be able to write code for parts of the client-side agent, while allowing instructors to monitor the learning progress of a student's agent due to the discounted cumulative rewards and the number of steps being synced between the client and server-side.

PNRL has been designed in such a way that it emphasizes the implementation of the Q-learning algorithm, and training the agent in learning to navigate around the puddles while maximize its discounted cumulative reward. PNRL is not concerned with saving the reinforcement learning model itself, or using this reinforcement learning model in different contexts. This allows game-based learning from building parts of the agent to happen effectively [6–8] by focusing students' learning onto the specific topics of learning and implementing an RL algorithm in an RL environment. This targeted learning objective of the system can make the learning curve for students learning of RL concepts a little less steeper. The system also allows for multiple client instances to join the game room and train their agents concurrently, enhancing its usability.

3.2.3 System Control Flow

The PNRL system's control flow begins when the server-side SmartFoxServer server game extension registers a request handler to handle client-side requests. The user then logs into the game with a username and password from the client-side and joins the game room. The client then sends a request to the server to provide the initial state of the agent, after which the object representing the user is added to a hash map storing key-value pairs of the users and their usernames on the server-side (to manage concurrent client instances). The server then responds with the initial state that the agent is in, and the client-side requests possible actions and rewards for those actions from the current state. The server validates this and responds with the possible actions and rewards from the current state. The client-side then uses the Q-learning algorithm with an epsilon-greedy policy to pick either the action with the maximum Q-value or a random action and sends the action to the server after updating its internal representation. The computed q and v-values in the Q and V tables on the client-side are updated for the current state, and state-action pair chosen. These are then sent over to the server for updating the master Q and V tables for monitoring. The number of training steps and discounted cumulative reward for an episode are incremented on the client and server-side after a move is sent to the server. This sequence of steps is repeated till a training episode termination condition is hit. After a training episode ends, the server and client-sides reset their representations of the agent's cumulative discounted rewards and step counts to 0, and move the agent to back to state 0 in preparation for the next training

episode if there is one. Both the client and server-sides log summary messages for each training episode.

A diagram of the control flow of the system in chronological order (reading the text on the arrows from top to bottom) is as follows:

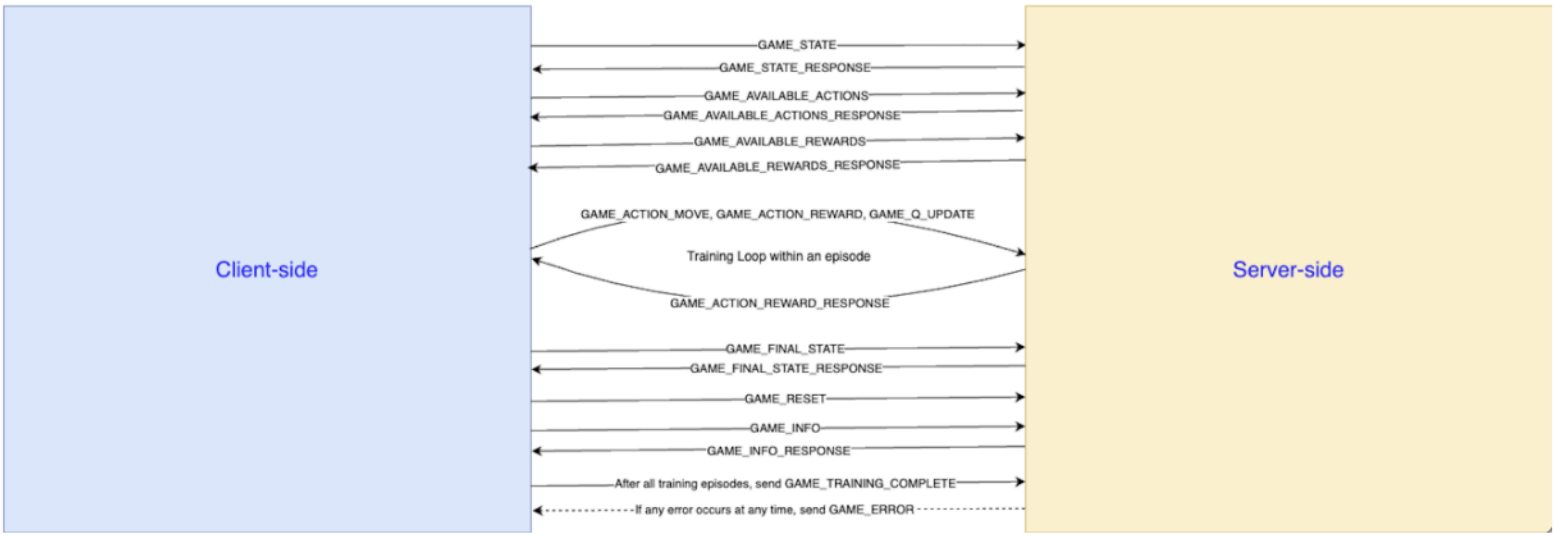

Figure 3.3: PNRL System Control Flow (Note: please zoom in to view the text in the diagram clearly or view figure 1 in Appendix A)

3.3 Training Termination Conditions

An overview and analysis of the agent's training termination conditions within PNRL are vital towards determining the training termination condition that is the most feasible for student learning of RL concepts for an undergraduate course in AI. The 3 training termination conditions with their descriptions are as follows:

3.3.1 Overview of Training Termination Conditions

1. Maximum steps training termination condition (STOP_METHOD=0): This training termination condition allows the agent to traverse the puddle world until the maximum number of steps (defined as a parameter within PNRL on the client and server-side) is reached for an episode. This condition is the default training termination condition for the system, and the training will keep going on until the maximum number of steps is reached or the goal state is reached, whichever comes first.
2. Goal state training termination condition (STOP_METHOD=1): This training termination condition allows the agent to traverse the puddle world until it reaches the goal state in an episode.
3. Probabilistic training termination condition (STOP_METHOD=2): This training termination condition allows the agent to traverse the puddle world until it is stopped by

the result of comparing whether a randomly generated probability is larger or smaller than the probability defined as a parameter to PNRL on the client and server-side (STOP_PROB) for each training step in every episode. If the random number generated is lesser than STOP_PROB, then the episode is terminated immediately, but if the random number generated is greater than STOP_PROB, then the agent's training process continues for one more training step. This process is repeated at the start of each agent step in the puddle world and training continues until either the termination condition ends the training episode, or if the RL agent reaches the goal state, whichever comes first.

3.3.2 Theoretical Analysis of Training Termination Conditions

Assuming the number of steps the agent traverses within a training episode as a measure of feasibility of the agent's training process of learning to navigate around puddle states (in terms of enabling students to train an agent that learns over a sufficient amount of training steps over a sufficient amount of episodes in a reasonable time frame), the probabilistic training termination condition can be modeled by a geometric probability distribution of a training episode terminating at exactly k steps within an episode. The parameter STOP_PROB acts as the parameter p for the geometric distribution probability mass function (PMF), whose equation is as follows [31]:

$$Pr(X = k) = (1 - p)^{k-1} p \tag{3.2}$$

The graph of the PMF of the agent's training termination at exactly k training steps as a function of $p = STOP\_PROB$ and k for the p values of $0.1, 0.3$, and 0.5 is as follows:

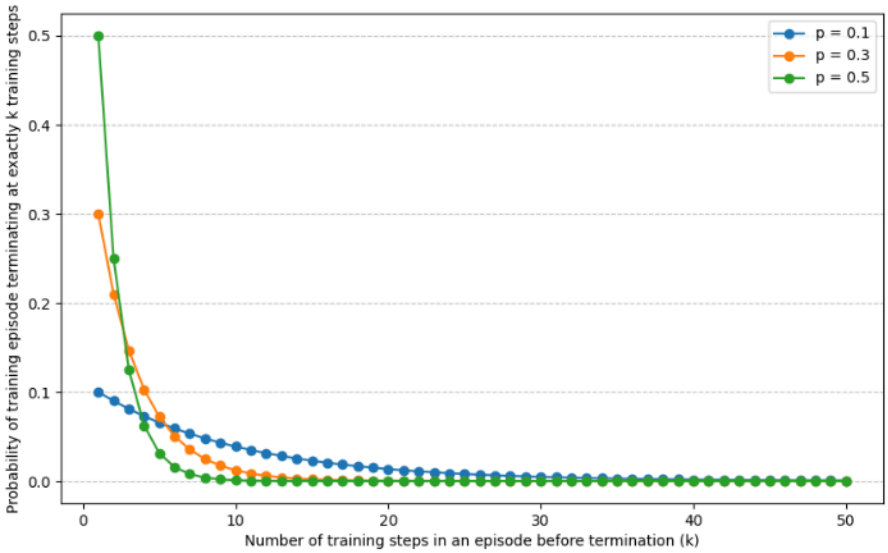

Figure 3.4: Geometric Distribution PMF of the number of training steps before terminating for different p values (Note: please zoom in to view the text in the diagram clearly)

From figure 3.4, one can deduce that the probability of the number of training steps within an episode being even more than a small value of 30 steps is quite low.

By definition, the probabilistic training termination condition's expected value of the number of steps that an episode will go on for before terminating is determined by the following formula for geometric distributions [31]:

$$E[X] = \frac{1}{p} \tag{3.3}$$

The expected values for the number of steps the agent trains for in an episode before the episode terminates computed using $p = 0.1, 0.3$, and 0.5 are 10, $3.33 \approx 3$, and 2 respectively. The number of steps in episode needs to be significantly higher than the values computed above to allow for any meaningful learning to happen within the agent over many training episodes. The larger the p value, the lesser experience that the agent is trained on, leading to ineffective balance of exploitation and exploration. Building an agent that does not train and learn for a sufficient amount of training steps and episodes is not helpful in aiding student learning of RL concepts. Therefore, this training termination condition can be theoretically deemed infeasible for aiding student learning of RL concepts in an AI course at the undergraduate level.

Using the above assumption of the number of steps the agent can traverse within a training episode acting as a measure of the agent's training feasibility, the goal state training termination condition is deterministic in terms of the condition it requires to terminate the training episode, but the probability of the agent reaching the goal state cannot be modeled as a distribution of training episode termination taking place at exactly k steps of training within an episode. This is because the learning process of the agent is determined by a combination of all factors mentioned in table 3.1, which influence the number of steps an agent takes to reach the goal state in different ways, causing heavy variability in training episode lengths. In particular, the grid size being set to even moderately large values like 9 or 10 makes the agent's learning process take extremely long to reach the goal state. Tagging on the additional layers of complexity introduced by having values of α, γ, and ϵ affecting the agent's learning process differently, along with the number of training episodes makes the training process become too time-intensive. This unpredictability in the number of steps and how long an agent takes to reach the goal state is not ideal in aiding student learning of RL concepts. Therefore, this training termination condition can be deemed theoretically infeasible for aiding student learning of RL concepts in an AI course at the undergraduate level.

Using the same assumption of the number of steps the agent can traverse within a training episode acting as a measure of feasibility of student learning of RL concepts, the maximum steps training termination condition can be modeled by a deterministic probability distribution of training terminating at exactly k steps within an episode. This distribution is known as the degenerate or dirac distribution [32]. The parameter k_0 is the value that the random variable X becomes with a probability of 1. The maximum number of steps (set as a parameter in PNRL, i.e. k) is set to k_0 for this training termination condition, guaranteeing training termination for an episode at exactly k_0 steps. The PMF for this distribution is as

follows [32]:

$$Pr(X = k) = \begin{cases} 1, & \text{if } \ x = k_0 \\ 0, & \text{elsewhere} \end{cases} \tag{3.4}$$

The graph of the PMF of agent training terminating at exactly k training steps as a function of $k = k_0 = MAX\_STEPS$ is as follows:

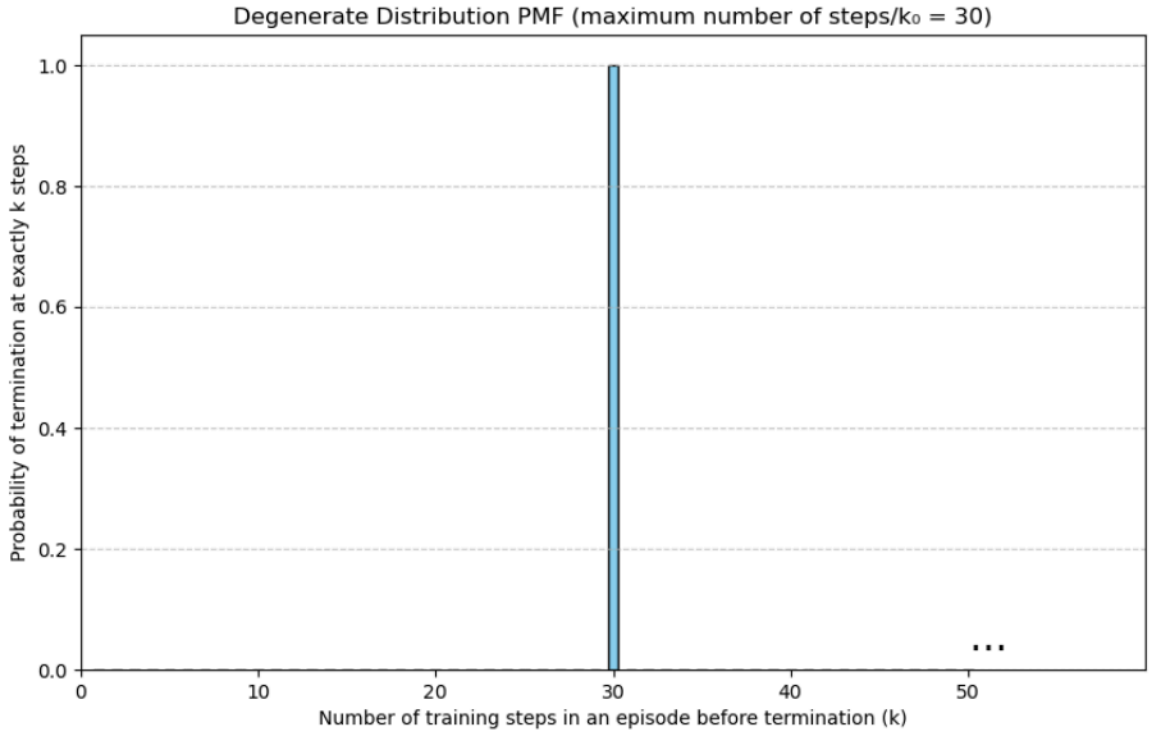

Figure 3.5: Dirac/Degenerate Distribution PMF of the number of training steps before terminating (Note: please zoom in to view the text in the diagram clearly)

From figure 3.5, the number of training steps per episode (k_0) was set to 30, and this number of training steps will occur with probability 1 for every training episode. By definition, the maximum number of steps training termination condition's expected value of the number of steps an episode will go on for before terminating is determined by the following formula for dirac distributions [32]:

$$E[X] = k_0 \tag{3.5}$$

Since the number of steps of the agent's traversal within an episode can be set based on preference and will always equal to k_0, the training process will not be time-intensive as the user can reasonably approximate how long the training process will take. This predictability

in the number of steps and how long an agent takes to reach the goal state is ideal in aiding student learning of RL concepts at the undergraduate level in an AI course, making it theoretically feasible to be used as the training termination condition within PNRL. This will allow the user to focus on building a deeper understanding of the agent's learning process due to the ability to customize the length of the training episodes.

3.4 Limitations

The limitation of PNRL is in relation to the grid size, puddle size and training termination condition used. For the maximum steps training termination condition, it is advised to set the number of puddles (MAX_PUDDLES) and the puddle size (PUDDLE_SIZE) to a small value assuming the grid size is a relatively small value (between 3 and 8). This grid size is suggested to ensure the agent can reach the goal state in a reasonable time frame during a training episode for either the maximum steps or the goal state training termination condition.

For example, if one were to set $MAX\_PUDDLES = 3$ and $PUDDLE\_SIZE = 3$ on $gridSize = 6$, the number of states that get covered with puddles is:

$$3 \text{ puddles} \times 3^2 \text{ states} = 27 \text{ states} \tag{3.6}$$

(out of total of 6^2 states on the grid), and will occupy majority of the state space of 36 states. This is detrimental towards the agent's learning process. Therefore, it is advised to either maintain MAX_PUDDLES in $1 - 3$ range in smaller grid sizes between 3 and 8. It is also advised to maintain:

$$MAX\_PUDDLES \times PUDDLE\_SIZE^2 \leq \frac{1}{4}\text{th of the total states in the grid} \tag{3.7}$$

as generating puddles that do not overlap with each other may not be possible under this configuration, completely blocking or covering a large part of the grid with puddle states. This limitation's relationship was deduced through trial and error while building the system. For the goal state training termination condition, it is advised to maintain the $gridSize$ value in the 1 to 8 range to reduce the time it takes for a training episode to complete, therefore, not indefinitely blocking the agent's learning process.

3.5 Python-Java Integration Architecture

The Python-Java code integration is separate from PNRL's architecture, and is built for allowing students to be able to program in a language that is more beginner-friendly, which is Python, and use its vast set of machine learning/reinforcement learning libraries to their advantage when programming the Game of Amazons project in UBC's COSC 322 AI course. The project is a two-player game for which the client-side is built by students with AI algorithms to play and win the Game of Amazons. The client-side uses SmartFoxServer's

Java client API to communicate moves to the COSC 322 game server (written with SmartFoxServer Java server API). The architecture of this integration consists of Python code written with Py4J, a python library allowing Python code to access Java objects through a JVM gateway, and placed directly into the client-side of the Game of Amazons. The integration code will call the necessary machine learning/reinforcement learning libraries, and perform the required computations to compute the best next move to be made. It will then communicate the move to be made in the game to the SmartFoxServer client API, which then communicates the move to make to the game server. A diagram with the data flow details of the integration with the COSC 322 UBC Game of Amazons client-server architecture is as follows:

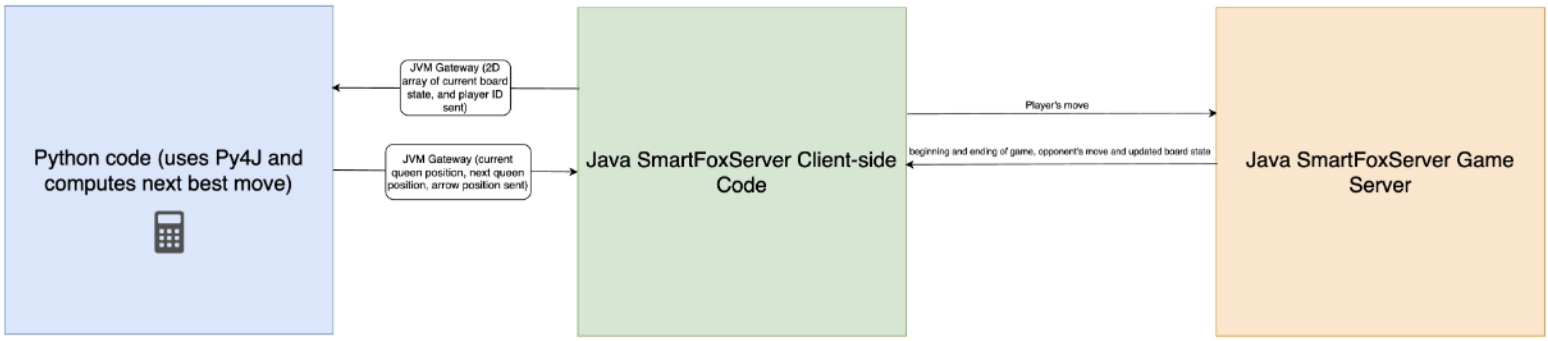

Figure 3.6: Python-Java integration architecture (Note: please zoom in to view the text in the diagram clearly or view figures 2 & 3 in Appendix A)

Having explained the PNRL's system architecture, communication protocols, client & server functions, control flow, completed a theoretical analysis of termination conditions, and stated the limitations of the system, the next chapter will test the effectiveness of PNRL as a tool in supporting student learning of RL concepts in an undergraduate AI course.

Chapter 4

PNRL Effectiveness Evaluation Results & Discussion

4.1 PNRL Assessment & Experiment Description

PNRL's agent and server-side must be thoroughly tested to gauge whether PNRL is a feasible option for supporting student learning of RL concepts in an undergraduate AI course. The research questions that data was collected and analyzed for will be introduced in subsection 4.1.1, providing insight into what PNRL as a whole will generally be assessed based on. In subsection 4.1.2, the metrics for assessing the effectiveness of the system's communication reliability, the RL agent's learning along with finding the most feasible training termination condition will be defined, followed by descriptions of the experiments and data collection process. In section 4.2, the results of the experiment will be presented, and section 4.3 will interpret the results of the experiments in the context of supporting student learning of AI concepts in an undergraduate level course through game-based learning.

4.1.1 Research Questions

The following research questions will be answered over the course of this chapter:

1. Is PNRL's system reliable in handling multiple back and forth communication messages?
2. Is the client-side agent in PNRL effective in its learning process?
3. Which training termination condition is most feasible in supporting student learning of RL concepts for undergraduate level students in an AI course?

4.1.2 Metrics & Experiment Details

A complete code implementation of the PNRL system has been provided as part of the work in this thesis, and this implementation is what will be used to collect data and answer research questions 1 to 3.

Experiment 1

To answer research question 1 as part of experiment 1, the metrics that are most useful in assessing whether PNRL's communication between client and server-sides is error-free are: check for a mismatch in the number of client and server-side discounted cumulative rewards, check for a mismatch in the number of client and server-side step counts, and the number of server-side error messages - all verified and counted at the end of each training episode. The primary focus of this thesis was developing a system that can allow for instructors to monitor students' learning of RL concepts while allowing for a training time-wise scalable agent training process, which, in turn, can aid student learning through practical implementation of RL theory. These metrics are complex enough to capture that the students are correctly understanding, calculating and communicating their agent's learning (rewards and step counts) in the training process to the server-side. PNRL was run 5 times to collect the above mentioned metrics as data for experiment 1. It also measures whether the server-side, can handle all the messages sent from the client and respond appropriately without running into any communication errors. The default training termination condition was used for collecting the data for this experiment, along with a grid size of 5 (25 total puddle world states), a small number of maximum steps per training episode of 30, 5 training episodes, step size α of 0.1, discount factor γ of 0.9, and an epsilon decay factor of 0.995. The default state, puddle state, and goal state transition rewards were set to -0.02, -2.0, and $+20.0$ respectively. The number of puddle groups (grouped squares of size $PUDDLE\_SIZE^2$) set to 1, with the $PUDDLE\_SIZE$ being set to 2 (4 puddle states in total). These environment and learning parameters were picked at random to assess if the system is capable of handling back-and-forth communication, and not to assess the performance of the agent after tuning these parameters.

Experiment 2

To answer the second research question as part of experiment 2, the metrics that are most useful in assessing the client-side agent's performance and progression are the change in the maximum Q-value of state 0 and the rolling variance of change in maximum Q-value of state 0 both averaged over 5 runs of the program. A complete code implementation of an agent that implements the Q-learning algorithm correctly must be assessed on its learning performance and progression to provide the skeleton code for students to work on to set the expectation for what they should aim for with their agents. The change in maximum Q-value of state 0 is tracked as a metric to observe whether the agent is gaining confidence in its (epsilon-greedy) policy's decisions. This metric also assesses if the action-value function estimates ($q_\pi(s, a)$) of the discounted cumulative reward from following policy π from selecting action a from state s producing the highest Q-value are stabilizing, indicating that the agent's learning is converging to close to optimal values. The rolling variance of the change in maximum Q-value of state 0 is also tracked to verify whether the change in maximum Q-value of state 0 is fluctuating too much over 5 runs and if the agent's learning is still unstable. PNRL was run 5 times to collect the above mentioned metrics as data for experiment 2. The cumulative discounted reward was not used as a metric in assessing the agent's performance due to the focus of this thesis being on building a system that helps students in gaining a good grasp

on implementing RL theory in practice and not simply on aiming for higher discounted cumulative rewards. State 0 in a comparatively less complex environment, like the puddle world, is specifically used to compute the change in maximum Q-value and rolling variance of change in maximum Q-value because it is always the starting point for a training episode, making it time-efficient to track the learning progression of the agent from. The default training termination condition was used for collecting the data for this experiment, along with a grid size of 4 (16 total puddle world states), a maximum steps per training episode of 300, 500 training episodes, step size α of 0.005, discount factor γ of 0.9, and an epsilon decay factor of 0.9. The default state, puddle state, and goal state transition rewards set to -0.0001, -0.002, and $+40.0$ respectively. The number of puddle groups (grouped squares of size $PUDDLE\_SIZE^2$) set to 1, with the $PUDDLE\_SIZE$ being set to 2 (4 puddle states in total). These environment and learning parameters were tuned to optimize and assess performance of the agent in terms of its learning stability and progression.

Experiment 3

To answer research question 3 as part of experiment 3, the metrics that are most useful in assessing which training termination condition is most feasible in supporting student learning of RL concepts are the change in the maximum Q-value of state 0 averaged over 5 runs of the program, the rolling variance of change in maximum Q-value of state 0 averaged over 5 runs of the program, and the wall-clock time it takes for training episodes to complete in one run of the program - all measured for each training termination condition. PNRL was run 5 times to collect the above mentioned metrics as data for experiment 3. Due to the focus of this thesis being as stated above, finding the training termination condition that has a balance of being less time-intensive while providing the most predictability in the agent's training process length and learning stability is the condition that will be deemed the most feasible out of the three implemented training termination conditions: maximum steps, goal state, and probabilistic. Since the data on the maximum steps training termination condition's learning stability was collected as part of experiment 2, the same data will be reused in the comparison for time feasibility and agent learning stability with the goal state and probabilistic training termination conditions. The same environment and learning parameters used for the maximum steps training termination condition from experiment 2 were used for collecting the data for the goal state and probabilistic training termination conditions in this experiment for ease of comparison and reproducibility.

The threshold for considering a change in maximum Q-value at state 0 as being too large between successive training episodes will be 0.01. This threshold was selected to be larger than the default and puddle state rewards since changes in maximum Q-value influence how the agent will decide its next move and subsequently what states' rewards it will receive, but significantly lower than the goal state reward to ensure that this threshold is not so high that the maximum Q-values never reach it in a reasonable training time frame. The threshold for considering a change in rolling variance of change in maximum Q-values for state 0 as being too large between windows of 10 training episodes will be 0.001 due to the fact that variance values observed are typically smaller compared to the change values themselves.

The client and server logs for all experiments were collected from SmartFoxServer2X server

application and a client instance run from a terminal locally on a laptop. The data for experiment 1 was collected manually from the extracted logs as it had very few training episodes, episode step counts, and discounted cumulative rewards. The data for experiments 2 and 3 were collected using Python scripts that used regular expressions to search through the client and server logs and keep a track of the change in the maximum Q-value of state 0 between episodes, exported the tracked values to a CSV (Comma Separated Values) file, and graphed the visuals for it. The wall clock times of running each termination condition were measured manually through the server logs files. The datasets of client and server logs along with the collected CSV files of data are all available here.

4.2 Experiment Results

4.2.1 Experiment 1 - System Effectiveness

A table containing the data of the client and server-side discounted cumulative rewards after each training episode, whether there is a mismatch between the rewards at the end of an episode, the client and server-side step counts, whether there is mismatch at the end of an episode, and the number of server-side error messages over 5 full runs of PNRL is recorded in table 4.1 to assess the system's effectiveness in handling communication. This experiment used the default training termination condition with the environment and learning parameters mentioned in subsection 4.1.2. The original dataset has boolean valued columns in place of the mismatch columns, which has been translated into yes or no text and the values in the server-side and client-side cumulative reward column have been rounded to 3 decimal places.

Table 4.1: System Effectiveness Assessment Table

Run #	**Episode #**	**Client-side step count**	**Server-side step count**	**Client-side reward**	**Server-side reward**	**Step count mis-match?**	**Reward mis-match?**	**Server error count**
1	1	22	22	-5.092	-5.092	No	No	0
1	2	30	30	-8.489	-8.489	No	No	0
1	3	30	30	-2.822	-2.822	No	No	0
1	4	30	30	-0.192	-0.192	No	No	0
1	5	14	14	4.935	4.935	No	No	0
2	1	30	30	-2.771	-2.771	No	No	0
2	2	19	19	-1.040	-1.040	No	No	0
2	3	30	30	-2.112	-2.112	No	No	0
2	4	30	30	-0.192	-0.192	No	No	0
2	5	30	30	-3.577	-3.577	No	No	0
3	1	30	30	-1.365	-1.365	No	No	0
3	2	30	30	-2.475	-2.475	No	No	0
3	3	30	30	-2.270	-2.270	No	No	0

Run #	Episode #	Client-side step count	Server-side step count	Client-side reward	Server-side reward	Step count mis-match?	Reward mis-match?	Server error count
3	4	26	26	1.250	1.250	No	No	0
3	5	30	30	-0.772	-0.772	No	No	0
4	1	30	30	-2.390	-2.390	No	No	0
4	2	30	30	-3.457	-3.457	No	No	0
4	3	30	30	-4.588	-4.588	No	No	0
4	4	24	24	-1.184	-1.184	No	No	0
4	5	30	30	-11.850	-11.850	No	No	0
5	1	10	10	7.626	7.626	No	No	0
5	2	30	30	-3.239	-3.239	No	No	0
5	3	30	30	-0.285	-0.285	No	No	0
5	4	30	30	-10.357	-10.357	No	No	0
5	5	24	24	1.183	1.183	No	No	0

4.2.2 Experiment 2 - Agent Effectiveness

The change in the maximum Q-value of state 0 (along with the moving average of the change in maximum Q-value of state 0) and the rolling variance of change in maximum Q-value of state 0 averaged over 5 full runs of PNRL using the default training termination condition and the environment and learning parameters mentioned in subsection 4.1.2 are depicted in figures 4.1 (zoomed in figure 4 in Appendix A) and 4.2 respectively.

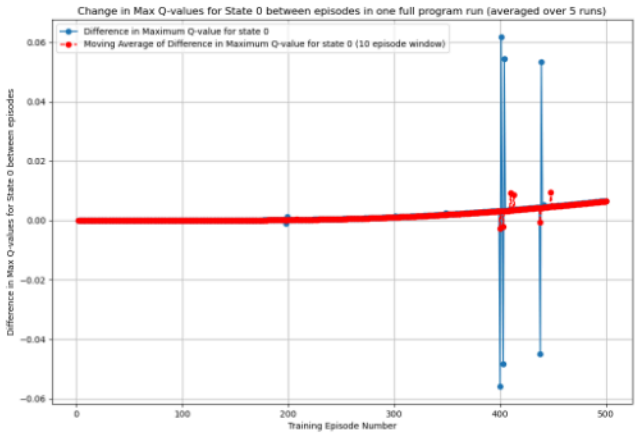

Figure 4.1: Change in maximum Q-value for state 0 averaged over 5 runs of PNRL (Note: please zoom in to view the diagram clearly)

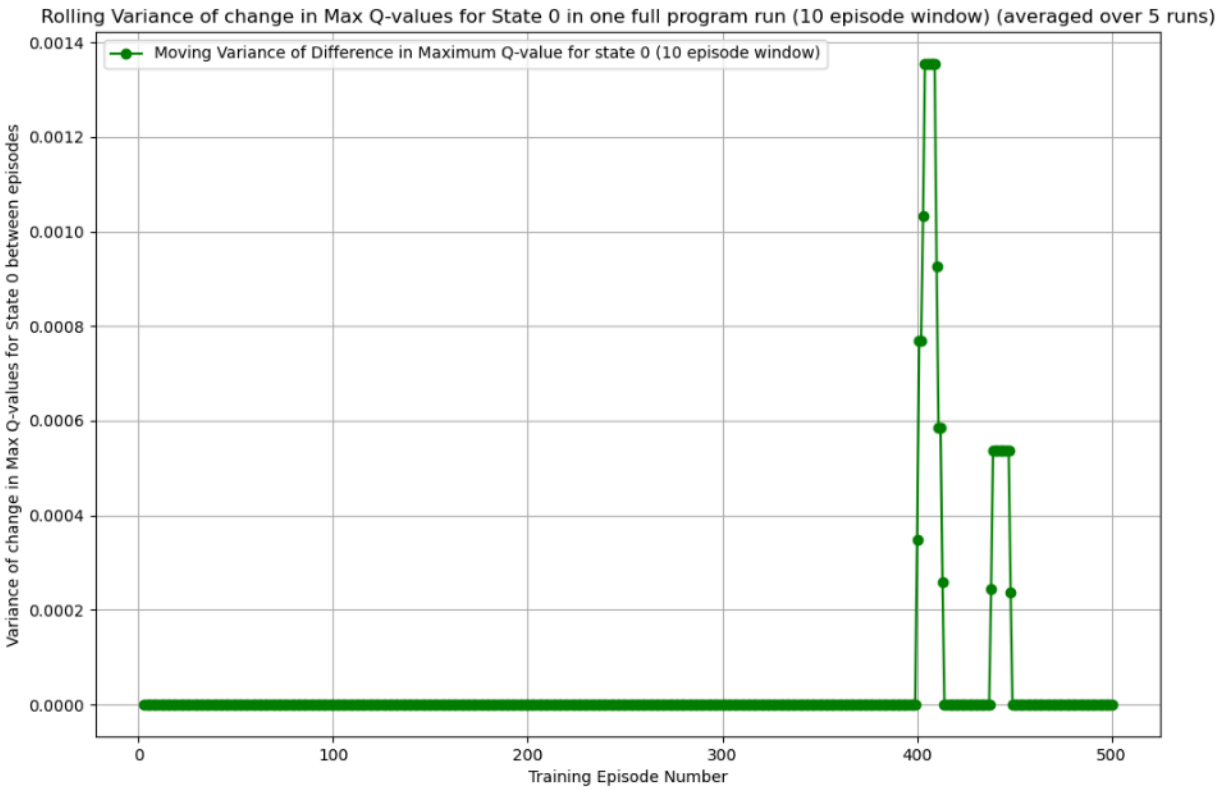

Figure 4.2: Rolling variance of change in maximum Q-value for state 0 averaged over 5 runs of PNRL (Note: please zoom in to view the diagram clearly)

4.2.3 Experiment 3 - Most Feasible Training Termination Condition

As the graphs of the change in maximum Q-value for state 0 and the rolling variance of change in maximum Q-value for state 0 using the maximum steps termination condition have already been generated for experiment 2 as figures 4.1 (zoomed in figure 4 in Appendix A) and 4.2, they will be referenced as part of experiment 3's discussion in section 4.3.

A table of the wall-clock times taken for executing a 500 episode, 300 steps per episode run of PNRL using the goal state termination condition with the above mentioned learning and environment parameters in subsection 4.1.2 is recorded in table 4.2.

The change in the maximum Q-value of state 0 (along with the moving average of the change in maximum Q-value of state 0) and the rolling variance of change in maximum Q-value of state 0 averaged over 5 full runs of PNRL using the goal state training termination condition with the above mentioned environment and learning parameters the mentioned in subsection 4.1.2 are depicted in figures 4.3 (zoomed in figure 5 in Appendix A) and 4.4 (zoomed in figure 6 in Appendix A) respectively.

Table 4.2: Maximum steps termination condition wall-clock execution time

Run #	**Wall clock time in seconds taken for training to complete**
1	55.255
2	50.681
3	50.083
4	47.022
5	49.360
Average: 50.480	

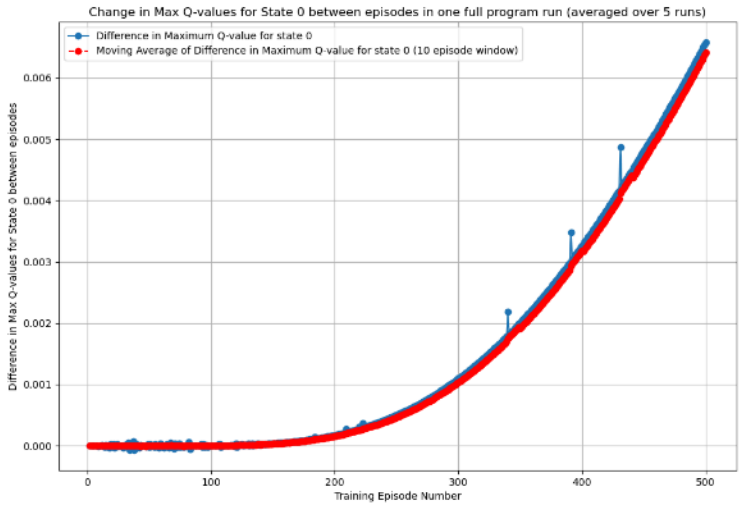

Figure 4.3: Change in maximum Q-value for state 0 averaged over 5 runs of PNRL for the goal state termination condition (Note: please zoom in to view the diagram clearly)

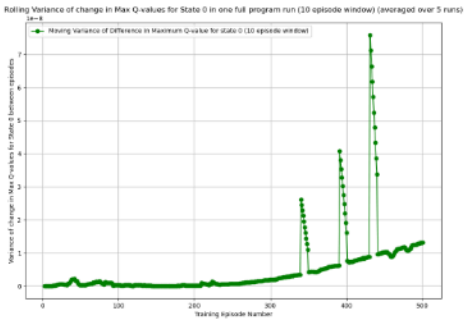

Figure 4.4: Rolling variance of change in maximum Q-value for state 0 averaged over 5 runs of PNRL for the goal state termination condition (Note: please zoom in to view the diagram clearly)

A table of the wall-clock times taken for executing a 500 episode, 300 steps per episode run of PNRL using the goal state termination condition with the above mentioned learning and environment parameters in subsection 4.1.2 is recorded in table 4.3.

Table 4.3: Goal state termination condition wall-clock execution time

Run #	**Wall clock time in seconds taken for training to complete**
1	47.726
2	46.373
3	60.887
4	44.577
5	47.776
Average: 49.468	

The change in the maximum Q-value of state 0 (along with the moving average of the change in maximum Q-value of state 0) and the rolling variance of change in maximum Q-value of state 0 averaged over 5 full runs of PNRL using the probabilistic training termination condition with the above mentioned environment and learning parameters parameters in subsection 4.1.2 are depicted in figures 4.5 (zoomed in figure 7 in Appendix A) and 4.6 (zoomed in figure 8 in Appendix A) respectively.

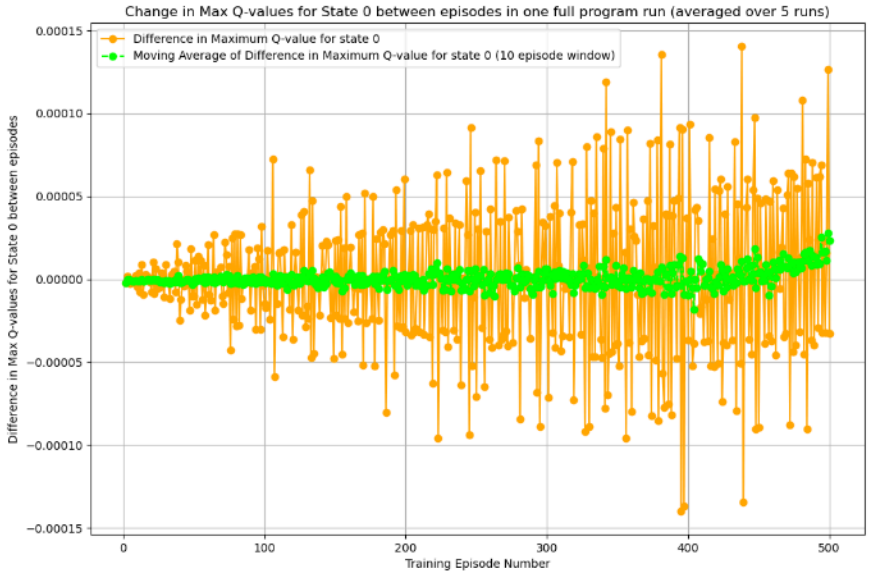

Figure 4.5: Change in maximum Q-value for state 0 averaged over 5 runs (with probabilities p = 0.1, 0.3, 0.5, 0.7 and 0.9) of PNRL for the probabilistic termination condition (Note: please zoom in to view the diagram clearly)

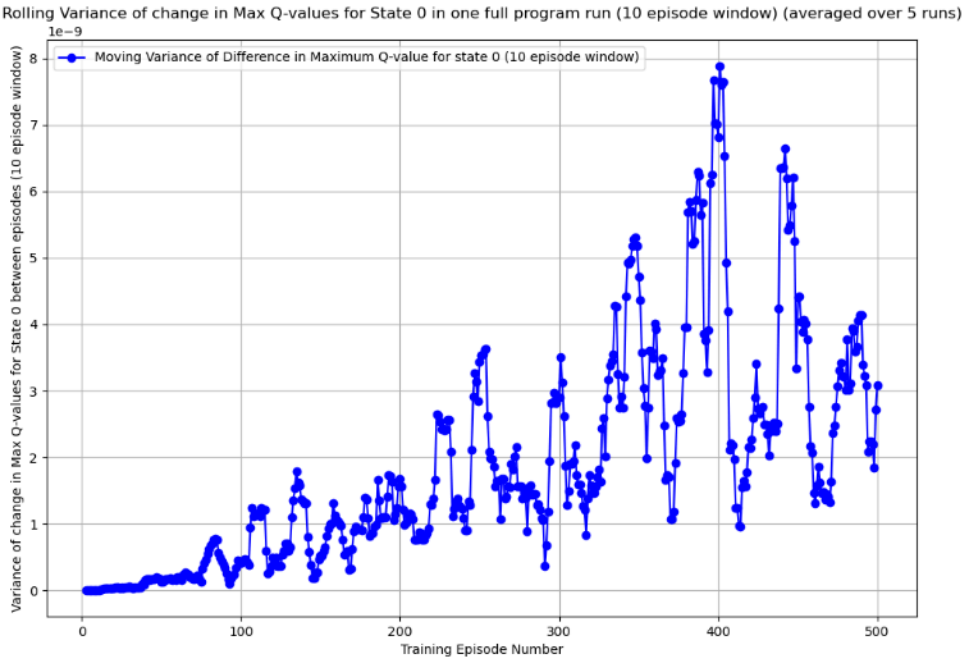

Figure 4.6: Rolling variance of change in maximum Q-value for state 0 averaged over 5 runs of PNRL for the probabilistic termination condition (Note: please zoom in to view the diagram clearly)

A table of the wall-clock times taken for executing a 500 episode, 300 steps per episode run of PNRL using the probabilistic termination condition with the above mentioned learning and environment parameters in subsection 4.1.2 is recorded in table 4.4.

Table 4.4: Probabilistic termination condition wall-clock execution time

Run #, probability of terminating training episode	**Wall clock time in seconds taken for training to complete**
1, 0.1	44.937
2, 0.3	22.256
3, 0.5	16.466
4, 0.7	13.760
5, 0.9	11.749
Average: 21.834	

4.3 Discussion

The following section will interpret the results from section 4.2 in the context of answering the 3 research questions proposed in section 4.1 and provide insights on why PNRL will be

effective as a teaching tool for RL concepts in an undergraduate AI course.

Experiment 1 Discussion

Subsection 4.2.1's table 4.1 demonstrates that the system does not produce any errors messages during its communication with the client. The table also shows that all the step counts and discounted cumulative rewards after each episode are in sync on the client and server-side. This indicates that the communication has been validated, no messages are being mishandled or data being lost during the communication in PNRL and therefore, the system as a whole can be considered reliable to be used for training RL agents in a puddle world environment.

Experiment 2 Discussion

The expectation of training an RL agent that uses Q-learning over many training episodes is that the change in maximum Q-values for state 0 will become very little eventually and converge to optimal values (q_*) with probability 1 [1]. Subsection 4.2.2's figure 4.1 depicts the change in maximum Q-value for state 0 for the agent staying at values very close to zero for majority of the training duration aside from rare spikes seen near episodes 400 and 450 which cross the threshold of 0.01 for change in maximum Q-values. This indicates that the learning process of the client-side agent remains stable and the agent gains confidence in its epsilon-greedy policy to exploit states with high Q-values. Although the Q-learning may not have fully converged to the exact optimal Q-value for state 0, this graph shows that the agent's training has come close to converging to the optimal Q-value for state 0, and can therefore, be considered an agent with an effective learning process in the puddle world environment. Figure 4.2 further reiterates the fact that the agent's learning of Q-values is coming very close to optimal Q-values from the variance of change of maximum Q-values for state 0 being significantly lower than the 0.001 threshold for majority of the training time with only rare spikes in the variance values, indicating learning convergence with little variability.

Experiment 3 Discussion

Observing the graphs for the goal state training termination conditions as seen in figures 4.3 and 4.4 reveals this condition's change in maximum Q-values for state 0 between episodes as increasing nearly exponentially towards the threshold of 0.01 as the number of episodes scales, with very little variability albeit an upward trend from the rolling variance graph (figure 4.4). This indicates that goal state termination has led to the agent's learning of the optimal Q-value for state 0 increasing over the training duration. The goal state training termination condition could be allowing the agent to train for too long and increasing exploration of suboptimal sets of steps in the puddle world. This training termination condition is not a scalable option for training an RL agent in a puddle world with a large amount of training episodes (which is usually required for the agent to achieve optimal performance in larger state spaces), due to the threshold of 0.01 for the change in maximum Q-values for state 0 being approached rapidly (figure 4.3). Another reason why this training termination condition is not a time-wise feasible option for students to use when testing out

their agents is that the wall-clock time it takes on average for a full run of PNRL to complete training (assuming the same environment parameters mentioned in 4.1.2, more specifically using a highly conservative 4 × 4 grid world size) is 49.468 seconds as recorded in table 4.3. If the grid size were to be increased even slightly, this wall-clock time will increase significantly. Therefore, this training termination condition is not feasible for supporting student learning of RL concepts in an undergraduate AI course.

Observing the graphs for the probabilistic training termination condition as seen in figures 4.5 and 4.6 reveals this condition's change in maximum Q-values for state 0 between episodes as fluctuating excessively but well below the threshold of 0.01. The rolling variance figure (figure 4.6) also indicates heavy fluctuation in the rolling variance of the change in maximum Q-values for state 0, but is well below the threshold of 0.001. This indicates that the agent's learning is likely insufficient even though both graphs are quite stable, due to the major issue of the wall clock-time that the agent trains for being very low on average across all probabilities. When the $STOP\_PROB$ parameter is set to lower values such as 0.1 it leads to nearly identical agent training wall-clock times to the maximum steps termination condition. But as we increase that probability, the training time drops significantly as can be seen from the values recorded in table 4.4. Keeping environment parameters the same as defined above, on average over all $STOP\_PROB$ values of 0.1, 0.3, 0.5, 0.7, and 0.9, the wall-clock training time is only 21.834 seconds, which does not enable the agent to learn and adjust Q-values of states for long enough before it begins to make optimal decisions in attempting reaching the goal state. This will inevitably lead to poor agent performance even after the number of training episodes is scaled up, due to the probability of stopping the training for an episode at exactly k steps on average being defined by the geometric distribution (equation 3.3) with parameter $p = STOP\_PROB$. This training termination condition can stop agent training abruptly leading to unpredictable outcomes, ineffective learning, and does not aid student learning of RL concepts as the agent will simply not train for long enough for students to see any meaningful progression in the agent's learning through the metrics defined. Therefore, this training termination condition is not feasible for supporting student learning.

The maximum steps training termination condition was deemed feasible in terms of its stability in change of maximum Q-values for state 0 between episodes and low rolling variability in change of maximum Q-values for state 0 from the experiment 2 discussion subsection above. The wall-clock times for the training process to complete under this condition on average are longer than the rest, sitting at 50.480 seconds. However, the advantage of using this training termination condition is that the probability distribution of the number of steps k that a training episode will execute for before terminating is modeled by the dirac distribution as mentioned in section 3.5 and is fully deterministic. The number of training steps in an episode can be controlled entirely by the MAX_STEPS parameter passed as an environment variable to the agent. This provides reasonably predictable behavior in terms of the agent's learning process and how long the training process takes in wall-clock time compared to the other two conditions. Due to this predictability, this condition's training time also scales well for larger grid sizes, and makes teaching RL concepts with it time-efficient. Therefore, the training termination condition that is the most feasible in supporting student learning of RL concepts in an undergraduate level AI course is the maximum steps training termination condition.

Students building parts of the client-side agent in PNRL focuses their learning on to a specific algorithm of how the agents learn, gain confidence in their policy, and how that policy can potentially converge to optimal values for the action-value (Q) function while allowing the instructors to monitor and assess a student's agent learning progression. Building the core of the agent's decision-making process and observing the interaction of the agent in puddle world through client logs constituting a game-based-learning-like environment also ensures that the students' base understanding of RL theory is strong, practical, and that they can take what they learn and apply it to optimize agent performance in not just a puddle world context, but even for different RL algorithms in different state spaces. PNRL serves as an efficient game-based-learning tool which can be effective in increasing students' understanding [6–8] of RL algorithms like Q-learning compared to having them solely study RL theory of building agents that aim to maximizing discounted cumulative reward without implementing these algorithms in practice. Therefore, the feasibility of PNRL with a maximum steps training termination condition in supporting student learning of RL concepts is high as it teaches the fundamentals of RL at a difficulty level that is neither too shallow nor too complex.

Chapter 5

Conclusion

This thesis has introduced a system that supports student learning of RL concepts in an undergraduate level AI course, called PNRL. It has also briefly described a Python to Java code integration for the Game of Amazons project for UBC's COSC 322 AI course. The internal client-server architecture of PNRL, its control flow, a theoretical analysis of training termination conditions within the system, and the system's limitations have also been described in detail. Three research questions were posed to thoroughly evaluate the system's client and server-sides, with the first question focusing on evaluating the reliability of the server and client-side's communication error-proneness, the second question focusing on evaluating the learning stability of the client-side agent, and the third question focusing determining the most feasible agent training termination condition in PNRL out of the three possible conditions implemented.

The first experiment evaluated the effectiveness of the server and client-sides of PNRL in handling multiple communications through measuring the accuracy of reward and step count syncing along with assessing the server error count. The second experiment evaluated the effectiveness of the client-side agent built to learn navigation of the the puddle world. This was done via assessing the stability in learning through measuring changes in the maximum Q-value changes between episodes for state 0, and the rolling variance of the maximum change in Q-value for state 0 over a 10-episode window. The third experiment determined the agent training termination condition that struck a balance of being least time-intensive while providing the most predictability in the agent's training process length and learning stability. The results of the first experiment proved that both the client and server-sides of PNRL are highly reliable in handling all communication that makes use of the communication protocol defined in subsection 3.2.1. The results of the second experiment proved that the client-side agent implemented in PNRL achieves learning stability in terms of the metrics defined, and can be provided to students with some methods filled out as the skeleton code for them to complete. The results of the third experiment proved that the maximum steps training termination condition had the best balance of not being least time-intensive while providing the most predictability in the agent's training time and learning stability, making it the most feasible to be used in PNRL to support student learning of RL concepts. The maximum training steps termination condition simplifies instructor evaluation and focuses

students' learning on the Q-learning algorithm rather than having to worry about how long training the model can take.

Future work on the PNRL system can involve extending it to save and load different RL models to experiment with in a puddle world setting. It can also be extended to integrate support for other RL, ML, and deep learning (DL) algorithms that increase the agent's performance in navigating the puddle world. Since PNRL was built to fit into an AI course's curriculum, integrating other types of learning (ML, DL) algorithms such as Deep Q-learning [33] into PNRL can make it a more holistic component to add into an AI course at the undergraduate level, encapsulating many different fields of AI.

Bibliography

[1] Sutton, Richard S. *Reinforcement learning: An introduction*. A Bradford Book, 2018. → pages 1, 2, 3, 4, 5, 6, 47, 48, 49, 50, 53, 54, 55, 57, 58, 59, 62, 63, 65, 66, 67, 73, 74, 75, 76, 78, 79, 80, 82, 83, 87, 88, 100, 119, 120, 121, 124, 128, 129, 130, 131

[2] Barto, Andrew G., Thomas, Philip S., and Sutton, Richard S. *Some recent applications of reinforcement learning*. In: *Proceedings of the Eighteenth Yale Workshop on Adaptive and Learning Systems*, 2017. → pages 5

[3] Alam, Ashraf. *A digital game based learning approach for effective curriculum transaction for teaching-learning of artificial intelligence and machine learning*. In: *Proceedings of the 2022 International Conference on Sustainable Computing and Data Communication Systems (ICSCDS)*, 2022, pp. 69–74. → pages 2

[4] Buss, Ray R., Wetzel, Keith, Foulger, Teresa S., and Lindsey, LeeAnn. *Preparing teachers to integrate technology into K–12 instruction: Comparing a stand-alone technology course with a technology-infused approach. Journal of Digital Learning in Teacher Education*, vol. 31, no. 4, pp. 160–172, 2015. → pages 14

[5] Voulgari, Iro, Marvin Zammit, Elias Stouraitis, Antonios Liapis, and Georgios Yannakakis. *Learn to machine learn: designing a game based approach for teaching machine learning to primary and secondary education students*. In: *Proceedings of the 20th Annual ACM Interaction Design and Children Conference*, 2021, pp. 593–598. → pages 1, 2, 3, 4, 5

[6] Zhan, Zehui, Yao Tong, Xixin Lan, and Baichang Zhong. *A systematic literature review of game-based learning in Artificial Intelligence education. Interactive Learning Environments* 32, no. 3 (2024): 1137–1158. → pages 2, 18

[7] Kazimoglu, C., M. Kiernan, L. Bacon, and L. Mackinnon. *A serious game for developing computational thinking and learning introductory computer programming. Procedia - Social and Behavioral Sciences* 47 (2012): 1991–1999. → pages 6, 7

[8] Mathrani, A., S. Christian, and A. Ponder-Sutton. *PlayIT: Game based learning approach for teaching programming concepts. Educational Technology & Society* 19, no. 2 (2016): 5–17. → pages 11, 12

[9] Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling. *The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research* 47 (2013): 253–279. → pages 2, 3, 4, 5, 6, 7, 8

[10] Johnson, Matthew, Katja Hofmann, Tim Hutton, and David Bignell. *The Malmo Platform for Artificial Intelligence Experimentation.* In: *IJCAI* 16 (2016): 4246–4247. → pages 1, 2

[11] Whiteson, Shimon Azariah. *Adaptive Representations for Reinforcement Learning.* The University of Texas at Austin, 2007. → pages 94, 95

[12] Bissyandé, Tegawendé F., Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. *Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects.* In: *2013 IEEE 37th Annual Computer Software and Applications Conference* (2013): 303–312. → pages 4

[13] Xu, Yongjun, Xin Liu, Xin Cao, Changping Huang, Enke Liu, Sen Qian, Xingchen Liu, Yanjun Wu, Fengliang Dong, Cheng-Wei Qiu, et al. *Artificial intelligence: A powerful paradigm for scientific research. The Innovation* 2, no. 4 (2021). Elsevier.

[14] McKinsey Global Institute. *Jobs Lost, Jobs Gained: What the Future of Work Will Mean for Jobs, Skills, and Wages.* (December 2017). [Online]. Available: `https://www.mckinsey.com/~/media/mckinsey/industries/public%20and%20social%20sector/our%20insights/what%20the%20future%20of%20work%20will%20mean%20for%20jobs%20skills%20and%20wages/mgi%20jobs%20lost-jobs%20gained_report_december%202017.pdf`

[15] Goertzel, Ben. *Artificial general intelligence: concept, state of the art, and future prospects. Journal of Artificial General Intelligence* 5, no. 1 (2014): 1. → pages 1, 2

[16] Kurzweil, Ray. *The singularity is near: When humans transcend biology.* Viking Penguin, 2005.

[17] Marrone, Rebecca, David Cropley, and Kelsey Medeiros. *How Does Narrow AI Impact Human Creativity? Creativity Research Journal* (2024): 1–11. → pages 1

[18] Shani, Lior, Yonathan Efroni, and Shie Mannor. *Exploration Conscious Reinforcement Learning Revisited.* In: *Proceedings of the 36th International Conference on Machine Learning* (2019): 5680–5689. → pages 1

[19] Laird, John E. and Robert E. Wray III. *Cognitive Architecture Requirements for Achieving AGI.* In: *Proceedings of the 3d Conference on Artificial General Intelligence (AGI-2010)* (2010): 3–8. → pages 1

[20] Adams, Sam, Itmar Arel, Joscha Bach, Robert Coop, Rod Furlan, Ben Goertzel, J. Storrs Hall, Alexei Samsonovich, Matthias Scheutz, Matthew Schlesinger, et al. *Mapping the Landscape of Human-Level Artificial General Intelligence. AI Magazine* 33, no. 1 (2012): 25–42. → pages 1

[21] Oracle. *Java SE 23 Documentation.* (n.d.). [Online]. Available: https://docs.oracle.com/en/Java/Javase/23/

[22] SmartFoxServer. *SmartFoxServer 2X API Documentation: Server.* (n.d.). [Online]. Available: https://docs2x.smartfoxserver.com/api-docs/Javadoc/server/overview-summary.html

[23] SmartFoxServer. *SmartFoxServer 2X API Documentation: Client.* (n.d.). [Online]. Available: https://docs2x.smartfoxserver.com/api-docs/Javadoc/client/

[24] Zyl, Jason van and Redmond, Eric. *Maven documentation – Maven.* (2009, Aug). [Online]. Available: https://maven.apache.org/guides/index.html

[25] Ninia. *Jep: Java Embedded Python.* (n.d.). [Online]. Available: https://github.com/ninia/jep.

[26] Py4J. *Py4J.* (n.d.). [Online]. Available: https://www.py4j.org/

[27] Agarwal, Nitin. *Reinforcement Learning: Q-learning Introduction (Part 1).* (n.d.). [Online]. Available: https://www.linkedin.com/pulse/reinforcement-learning-Q-learning-introduction-part-1-nitin-agarwal/

[28] Stack Overflow. *Epsilon and learning rate decay in epsilon-greedy Q-learning.* (n.d.). [Online]. Available: https://stackoverflow.com/questions/53198503/epsilon-and-learning-rate-decay-in-epsilon-greedy-Q-learning

[29] Sutton, Richard S. *Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding.* In: *Advances in Neural Information Processing Systems* 8 (1995). → pages 1041

[30] LinkedIn. *What are the benefits and drawbacks of using decaying epsilon?* (n.d.). [Online]. Available: https://www.linkedin.com/advice/0/what-benefits-drawbacks-using-decaying-epsilon#:~:text=1%20Benefits%20of%20decaying%20epsilon,-A%20decaying%20epsilon&text=First%2C%20it%20can%20help%20the,confidence%20in%20its%20value%20function.

[31] Wikipedia. *Geometric distribution.* (n.d.). [Online]. Available: https://en.wikipedia.org/wiki/Geometric_distribution

[32] Wikipedia. *Degenerate distribution.* (n.d.). [Online]. Available: https://en.wikipedia.org/wiki/Degenerate_distribution

[33] Hugging Face. *Deep Q Algorithm.* (n.d.). [Online]. Available: https://huggingface.co/learn/deep-rl-course/en/unit3/deep-q-algorithm

Appendix

A Appendix A: Figures

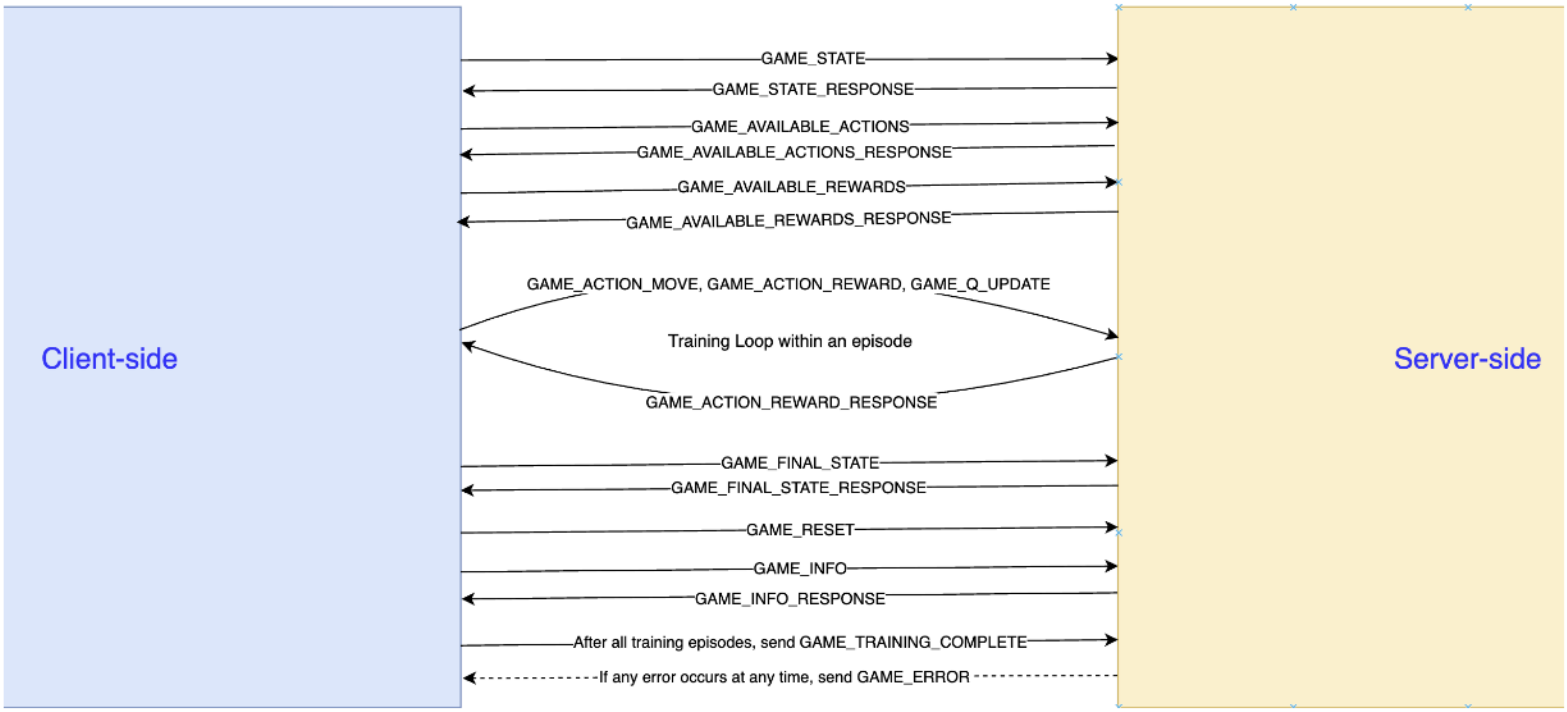

Figure 1: PNRL System Control Flow (zoomed in)

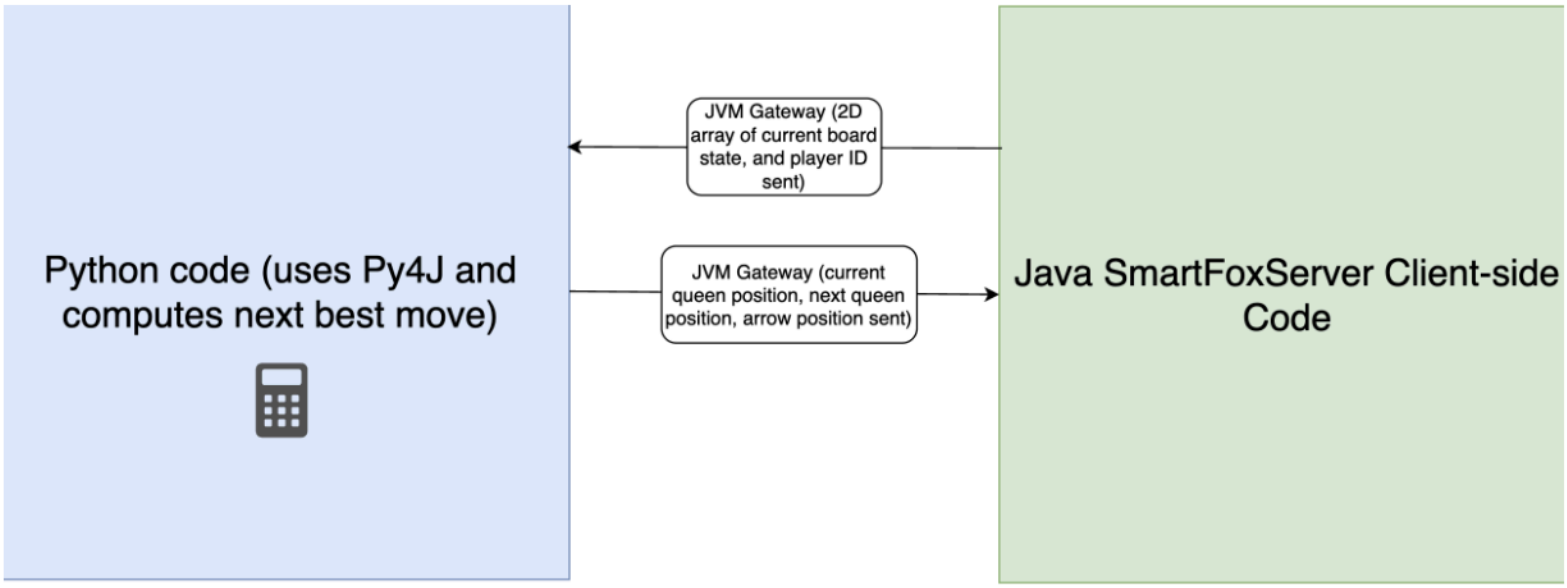

Figure 2: Python-Java client data flow (zoomed in)

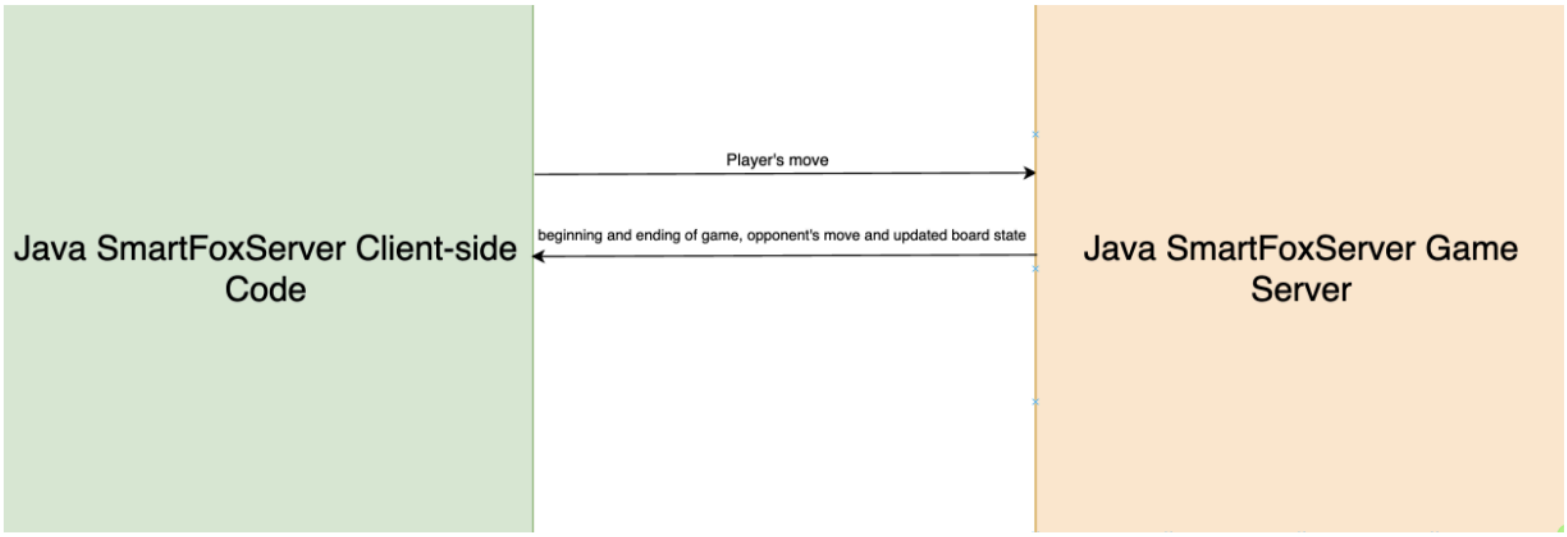

Figure 3: Java client and game server data flow (zoomed in)

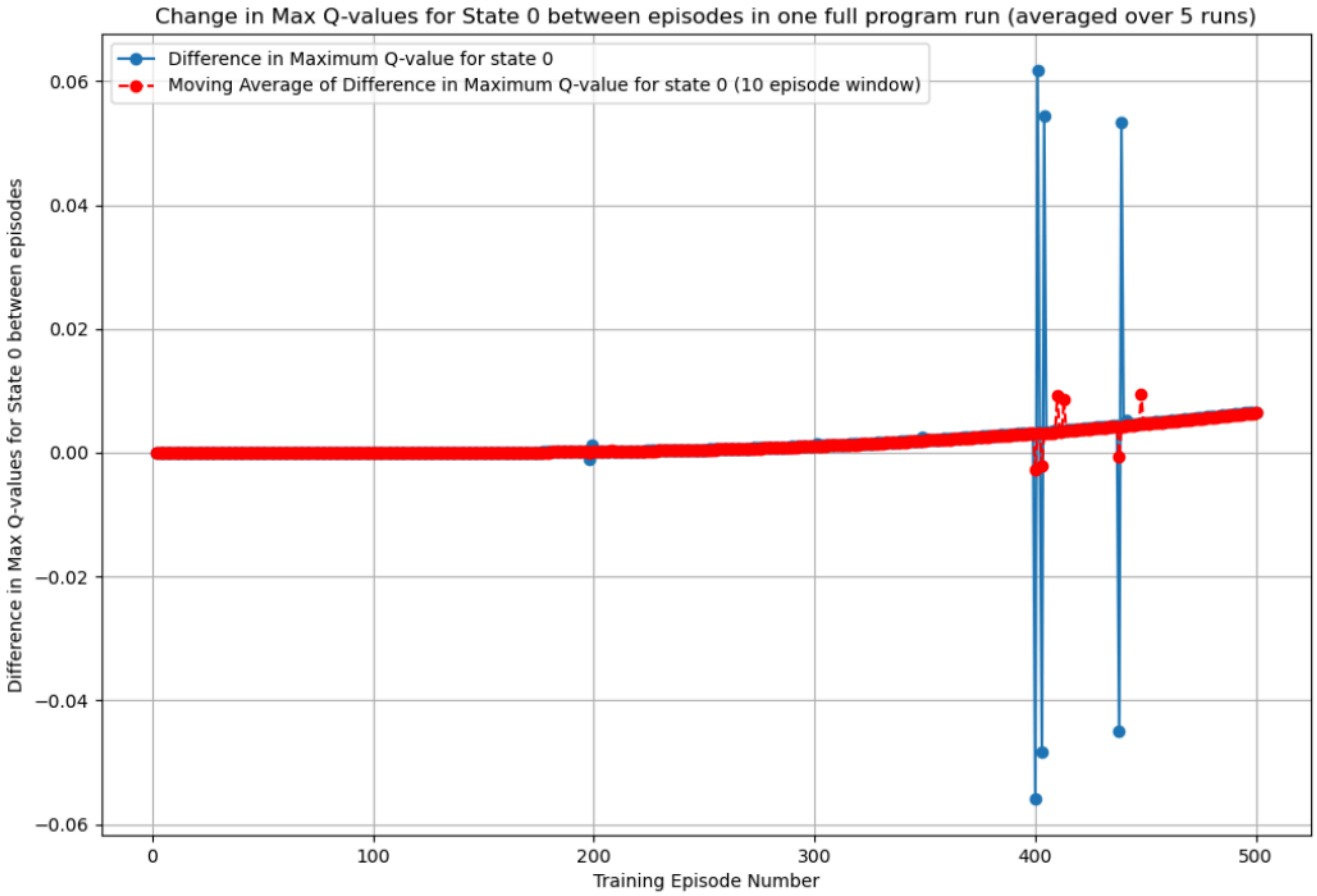

Figure 4: Change in maximum Q-value for state 0 averaged over 5 runs of PNRL (zoomed in)

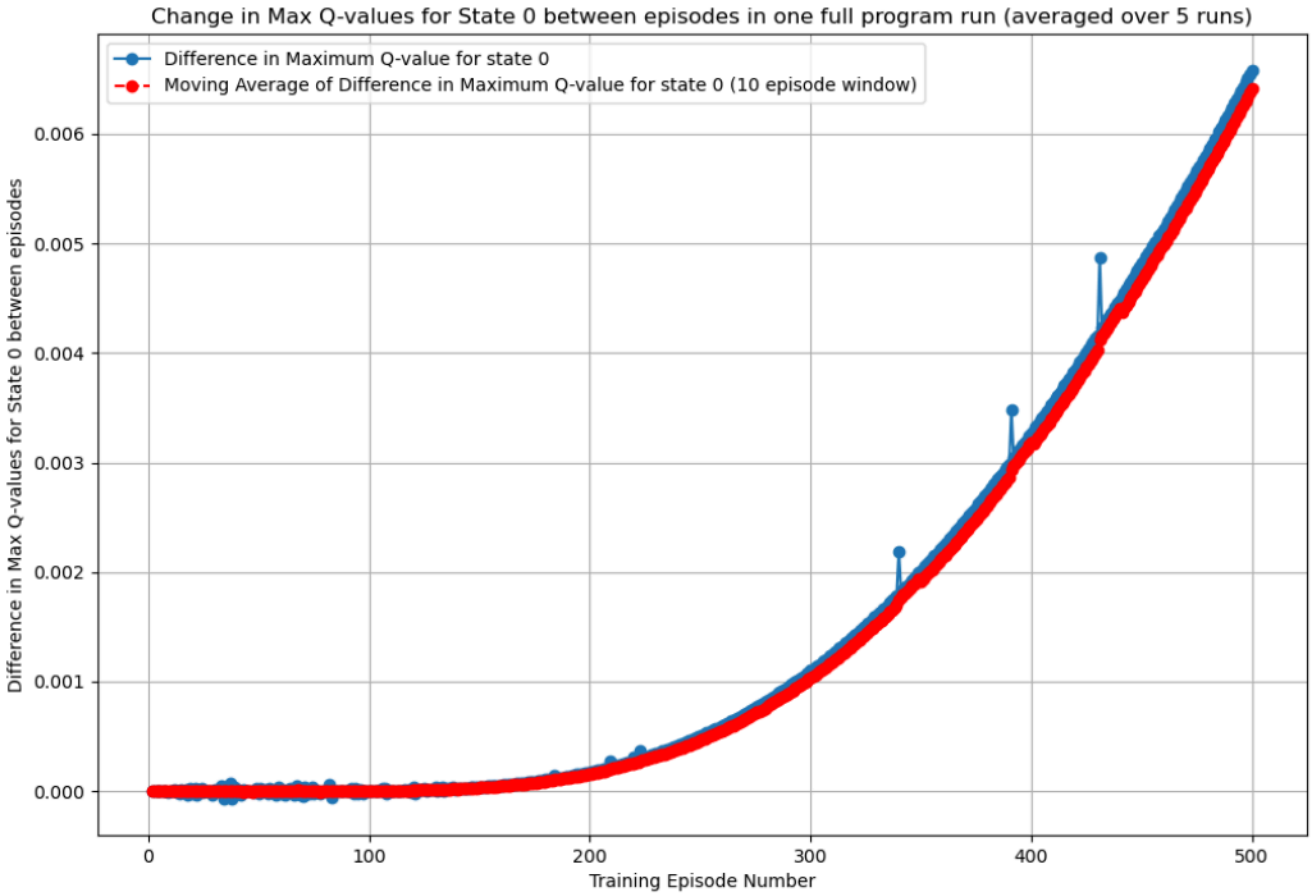

Figure 5: Change in maximum Q-value for state 0 averaged over 5 runs of PNRL for the goal state termination condition (zoomed in)

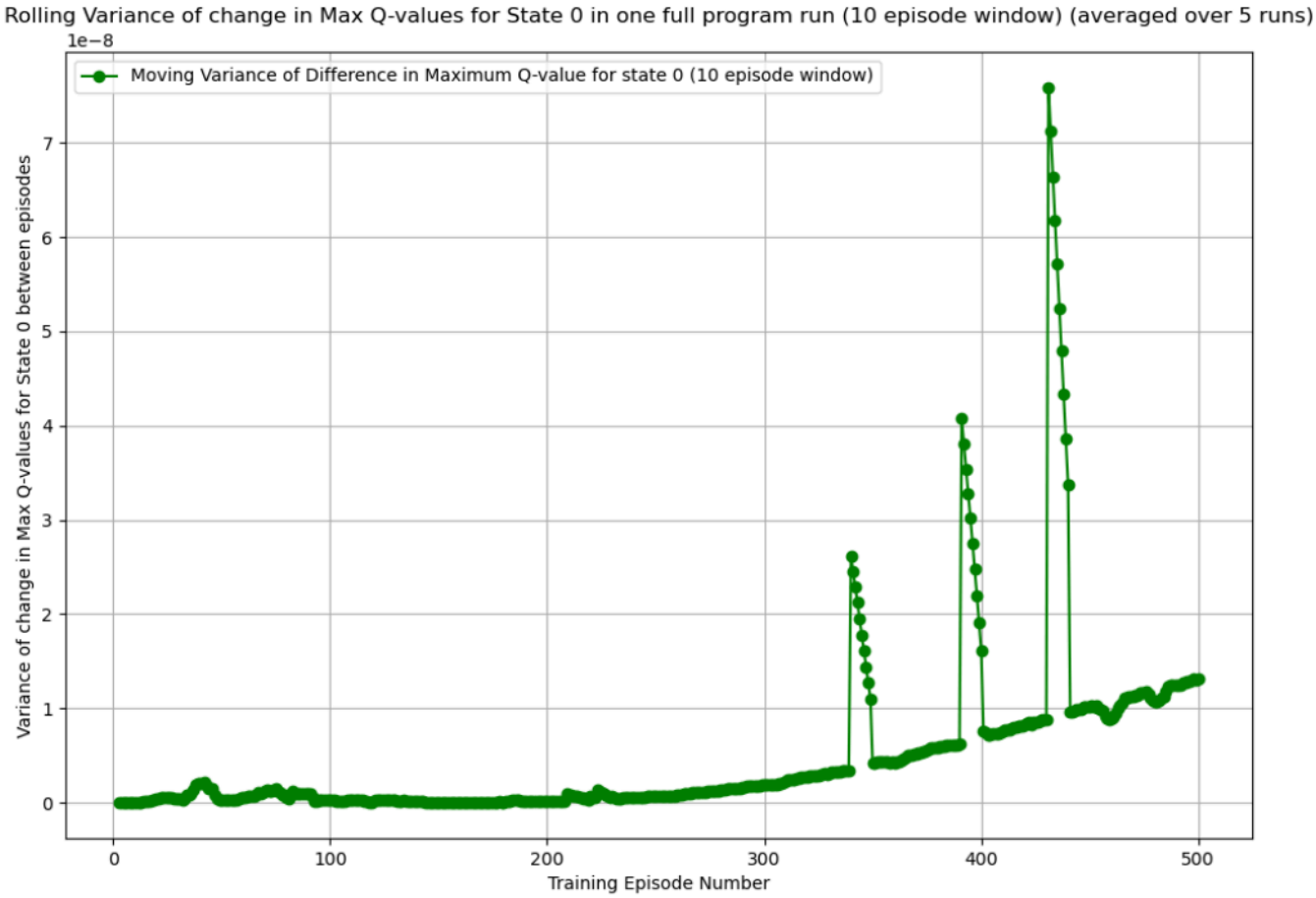

Figure 6: Rolling variance of change in maximum Q-value for state 0 averaged over 5 runs of PNRL for the goal state termination condition (zoomed in)

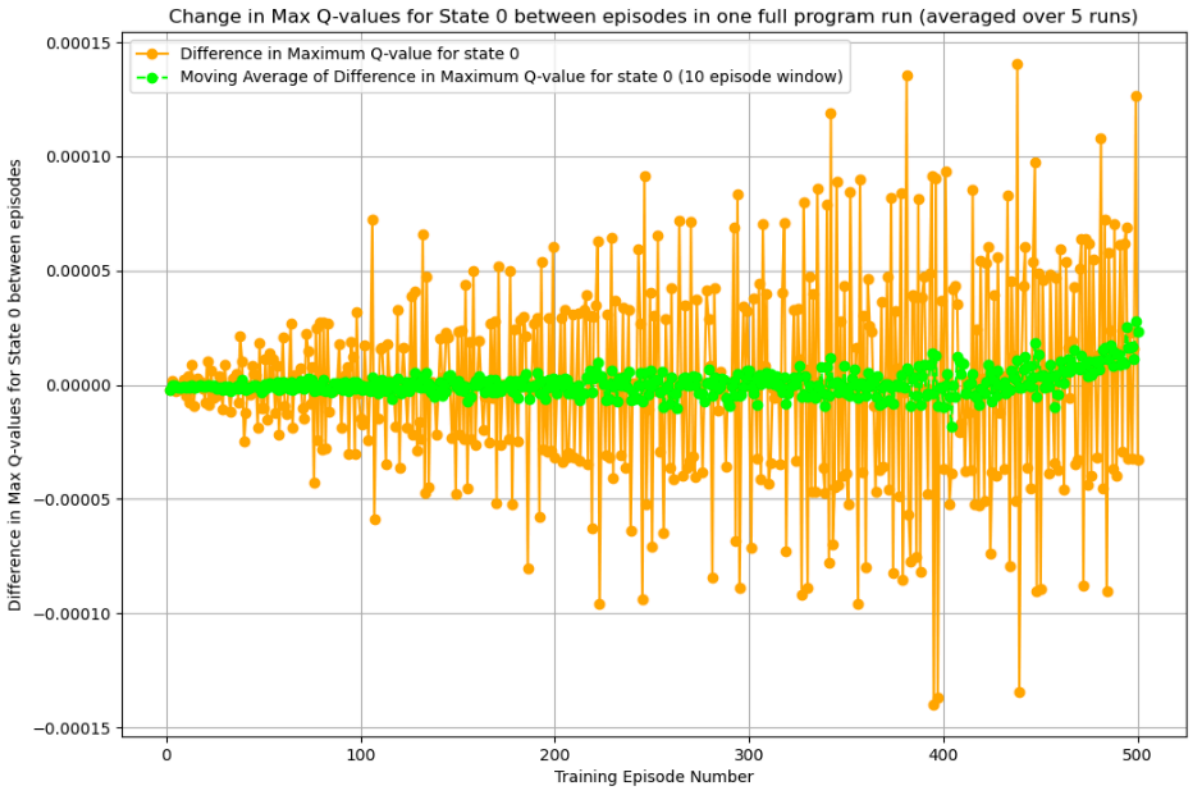

Figure 7: Change in maximum Q-value for state 0 averaged over 5 runs (with probabilities p = 0.1, 0.3, 0.5, 0.7 and 0.9) of PNRL for the probabilistic termination condition (Note: please zoom in to view the diagram clearly)

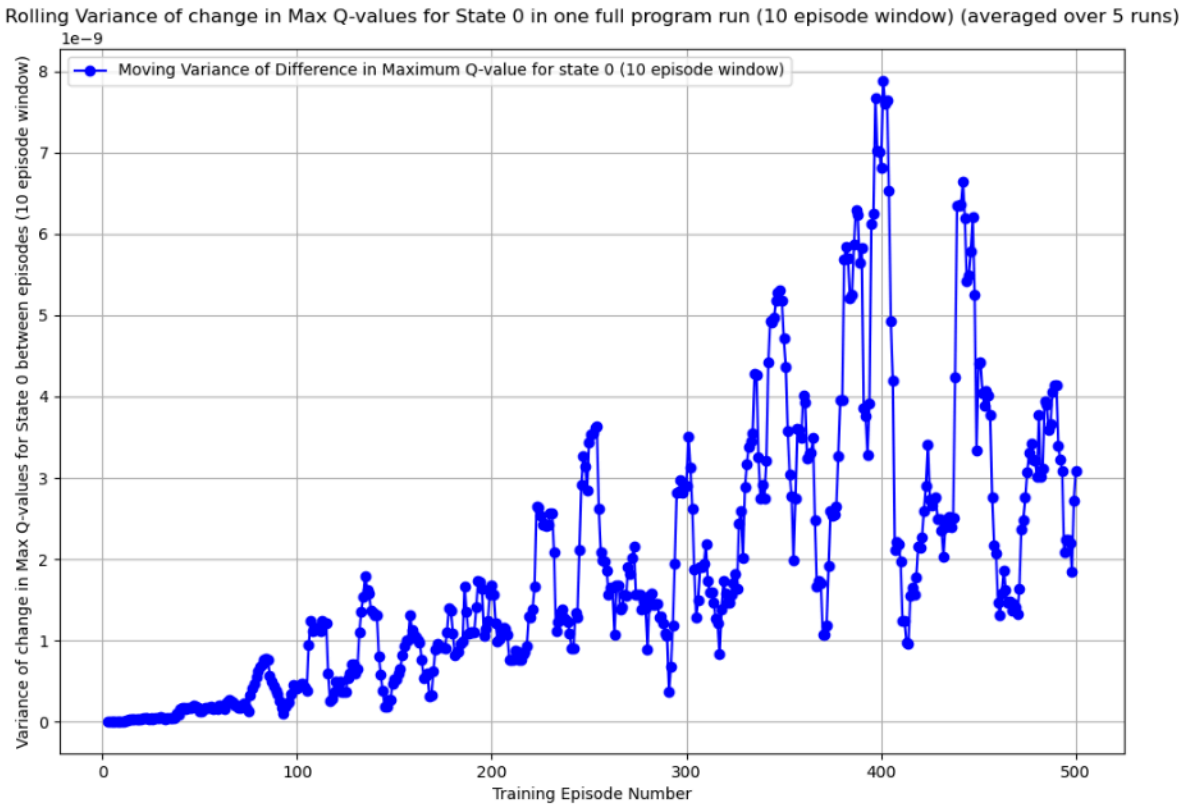

Figure 8: Rolling variance of change in maximum Q-value for state 0 averaged over 5 runs of PNRL for the probabilistic termination condition (zoomed in)

B Appendix B: Algorithms

Algorithm 4 Iterative Policy Evaluation to estimate the value function V via v_π

Input: Policy π for evaluation

Parameter: Small Threshold for $\theta > 0$ to determine the accuracy of the value estimated

Initialize $V(s)$ with arbitrary values $\forall s \in \{\mathbb{S}^+ - s_{terminal}\}$ (every state except the terminal state) and $V(s_{terminal}) = 0$

Loop:

$\Delta \leftarrow 0$

Loop for each state $s \in \mathbb{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$

$\Delta \leftarrow max(\Delta, |v - V(s)|)$

Until $\Delta < \theta$

Algorithm 5 Policy Iteration - evaluation and improvement to estimate the optimal policy π_* via π

1. Initialization of $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ with arbitrary values $\forall s \in \{\mathbb{S}^+ - s_{terminal}\}$ and $V(s_{terminal}) = 0$

2. Policy Evaluation

Loop:

$\Delta \leftarrow 0$

Loop for each state $s \in \mathbb{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$

$\Delta \leftarrow max(\Delta, |v - V(s)|)$

Until $\Delta < \theta$

3. Policy Improvement

$policyStable \leftarrow true$

For each $s \in \mathbb{S}$:

$oldAction \leftarrow \pi(s)$

$\pi(s) \leftarrow \underset{a}{argmax} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

If $oldAction \neq \pi(s)$, then $policyStable \leftarrow false$

If policyStable, then terminate and return $V \approx v_*$ and $\pi \approx \pi_*$, else go back to 2

Algorithm 6 Value Iteration - estimation of the optimal policy π_* via π

Parameter: Small Threshold for $\theta > 0$ to determine the accuracy of the value estimated

Initialize $V(s)$ with arbitrary values $\forall s \in \{\mathbb{S}^+ - s_{terminal}\}$ and $V(s_{terminal}) = 0$
Loop:
    $\Delta \leftarrow 0$
    Loop for each state $s \in \mathbb{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \underset{a}{max} \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$
        $\Delta \leftarrow max(\Delta, |v - V(s)|)$
Until $\Delta < \theta$