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 has also 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. 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 the web services [2].

While reinforcement learning has been used extensively in game-related applications, there is a lack of adoption of using the construction of reinforcement learning systems in a game-based learning context to support learning of RL concepts at the undergraduate level. This eludes students of seeing these concepts in action, leading to 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 emphasize on the AI concepts has previously been shown to increase the quality of teaching of AI 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 Malmo, a Minecraft-like platform for building and testing agents on various complex tasks in a continuous 3D environment [10].

A puddle world is a grid world containing puddles that produce a negative reward for a reinforcement learning agent that steps in it. 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 allowed to traverse within. 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 value of their reward function at the end of their training episodes. The game of Amazons is a dual player game where both players have to move their set of pawns to restrict the other players pawns from being able to move with the use of arrows that can be shot in a particular direction. 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 when in an undergraduate level programming course, are sometimes restricted to using a single programming language that they may not be familiar with. 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 the 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 [11], 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 PWRLSys (Puddle World Reinforcement Learning System), a client-server-based backend architecture for a game-based RL system. It includes a fully-functional RL agent within a puddle world, whose effectiveness in different environment settings is evaluated via learning metrics such as: cumulative reward per training episode, statistical tests for learning, and reward variance to demonstrate the effectivensss of the agent, and other system-related metrics are computed to verify the effectiveness of the system as a whole. 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 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 termination conditions for the RL agent to find the ideal settings to support student learning of the 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 a language that they may be more comfortable with.

Of the remaining chapters of this thesis, chapter 2 will focus on describing concepts of AI, notations for RL, the algorithms in RL, and will contain brief literature reviews of the existing platforms for RL agent development, and existing frameworks for integrating Python code with Java code. Chapter 3 will start by describing the structure and aspects of the puddle world in PWRLSys, the architecture of PWRLSys, and will end with a brief description of the architecture of the Python-Java integration for the Game of Amazons. Chapter 4 will go over experimenting the RL agent's effectiveness with different environment conditions, and traversal termination conditions, and 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 learnings to complete tasks [12]. As mentioned earlier, AI has become a bigger part of society and modern workflows in many subject domains with every growing year, and 75 to 375 million people will have their jobs replaced by Artificial Intelligence [13]. The goal of developing ”narrow AI” [14, 15] 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 [16]. Focusing on building systems that can reason efficiently in a variety of domains is to focus on constructing an Artificial General Intelligence (AGI) system. General intelligence has no unifying definition but consists of: being able to perform tasks in various different situations and surroundings, 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 [14].

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 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 [17], 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 [17], and focusing on exploiting actions whose outcomes are known aim to increase immediate rewards, and focusing on exploring new actions whose outcomes are unknown aim to increase future rewards in the environment. Algorithms that learn how to balance the exploration-exploitation trade-off tend to succeed in their 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 can generalize to different environments, or whether it simply memorized the training environment, leading to over-fitting. The upcoming sections will define relevant RL notation and concepts, and conduct a literature review of the existing RL platforms and Python-Java cross-integration platforms.

2.1 Introduction to Reinforcement Learning

2.1.1 Markov Decision Processes

To explain how decisions affect future rewards, and states that the agent could potentially enter, one requires a structure capable of holding this information for an RL agent in an environment, and this is what a Markov Decision Process (MDP) functions as. A state can represent any discrete part of an environment that an agent is present in, and an action can represent any task that an agent performs to move out of the current state. MDPs are central to understanding reinforcement learning and estimate $q_*(s, a)$, which is the total expected reward for an agent if it takes an action a in state s and makes the optimal decisions from the next step until the goal state or $v_*(s)$, which is the reward received from being in a state s and then following the optimal policy till the goal state [2]. The environment that the agent traverses through will be defined as discrete for the rest of this thesis, due to each action of the agent happening in a separate time step than the agent's previous action. The MDP leads to formulation of a specific type of order of states, actions, and rewards which can be defined as such [2]:

$$S_0, A_0, R_0, S_1, A_1, ... \tag{2.1}$$

[2]

In this thesis, the focus will be on a finite state-action-based puddle world, which is why it is important to establish that the relevant MDPs to focus on will be the discrete version with S, A, and R all have finite values in their relevant sets [2]. Discrete MDPs have discrete random variables S and R that hold the probability distributions that are entirely dependent on the previous states and actions performed by the agent from the previous state to get to the next state between time steps $t-1$ and t[2]. This property that MDPs exhibit is known as the Markov Property [2]. The probability of transitioning to state s' with reward r from a previous state s after performing action a is as follows [2]:

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

[2]

The sum of all these probabilities in this probability distribution sum up to 1, and equation 2.2 can determine the state of an RL agent at any time step, which is useful in understanding the behavior of the RL agent in a discrete environment, and leverage that understanding to build agents that can navigate different domain-specific environments. The exact expected rewards for any state-action-next-state triplets between time steps t and $t-1$ as summation over all possible rewards can be computed with the following formula [2]:

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

[2]

Equation 2.3 can also be utilized in a similar way to equation 2.2 to understand the behavior of the agent in scenarios involving different magnitudes of rewards, and can inform algorithm design on course correction if needed. A central assumption to MDPs is that the agent knows some information about its environment [2], and more knowledge on the environment is obtained through the agent's experience of reacting to its surrounding states and the rewards associated with those states in order to maximize the expected value of the cumulative reward obtained by the end of the agent's traversal of the environment [2]. 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 on the state s' it is in, through the reward assigned from being in that state. The expected final cumulative reward summed from time steps t (initial state) to T (final state) can be defined as follows [2]:

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

[2]

This formulation of the cumulative rewards fits perfectly into applications where a sequence of steps leads to a final state known as the terminal state, and this sequence of steps where the agent interacts with the environment is called an episode [2]. 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 [2]. Tasks that can be split into episodes are known as episodic tasks [2], and forms the basis of the puddle world explained in the next chapter. S represents all the non-terminal states in an episode and S^+ represents all the states including the terminal state [2]. 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, attempts to maximize the cumulative reward of the agent, which are known as continuing tasks [2].

Evaluating rewards from both long-term and short-term perspective is important to ensure an agent's effectiveness, and this is where the discount factor γ is vital [2]. The discount factor is a value between and including 0 and 1, which essentially handles this by decreasing the weight of rewards attained k time steps from the current time step by a factor of γ^{k-1} [2]. If one were to assume a γ value of close to 0, the agent would give a lot of weight towards seeking immediate rewards, and very little weight towards seeking future rewards from states that may not have been explored yet. And conversely, if one were to assume a γ value of close to 1, the agent would give a lot of weight towards seeking future rewards, and very little weight towards seeking immediate rewards [2].

Integrating the discount factor into the final cumulative reward calculation equation, the following formula can be deduced where the discount factor is multiplied k - 1 times to the reward at time step k and summed up with the rewards from the previous time steps [2]:

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

[2]

The value of G_t has a specific pattern that can be noticed from equation 2.5, and can be substituted to simplify the expression further to [2]:

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

[2]

The final cumulative reward equation can be unified to cover episodic and continuing tasks by replacing the regular terminal state with a state called absorbing state [2], and this state continues to add a reward of 0 forever in the case of an episodic task, 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:

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

[2]

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 cumulative rewards, and these functions are known as value functions [2]. An agent's policy, represented as π, is defined as pairings of probabilities of choosing a particular action to perform from a given state with that state, and forms the general strategy that the agent uses to navigate the environment it is in. The value function of state s under a policy π, represented by $v_\pi(s)$ is the expected final cumulative reward for an agent that starts in state s and follows the policy π from thereon [2]. The formulation of the value function is as follows [2]:

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

[2]

The value of a different function, known as the action-value function, represents the expected cumulative reward from following policy π and selecting action a from state s [2]. The formulation of the action-value function is as follows [2]:

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

[2]

These value and action-value functions are estimated from the sequence of interactions that the agent has with its environment, and the estimation of these functions can be done by averaging the value function's value for all the successive states from state s or $v_\pi(s)$, and tracking separate averages for each action taken from each state, these values will converge to the action-values from $q_\pi(s,a)$ [2]. This averaging method is known as monte carlo estimation, and is practical in the case where there are a limited number of states, such as in a puddle world [2]. Monte Carlo estimation enables the agent to learn directly from experience of interacting with environment without an understanding of the dynamics of the environment [2].

An important property of value functions is that they can be defined recursively in a similar way to how the cumulative reward function G_t [2]. The deduction of this recursive relationship comes from substituting equation 2.6 into the expected value calculation and considering the policy that an agent uses to pick the next state it transitions to and 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 by summing over the successive states and rewards [2]. The value of x from the $E[x]$ 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 [2]:

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

[2]

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' [2]. Essentially, the Bellman equation states that the value of the starting state s following a policy π is equal to the discounted next state value plus the reward to transition to that state $(r + \gamma v_\pi(s'))$, and this is averaged over all possible values for the upcoming states multiplied by the probabilities of those values occurring $(\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a))$[2]. The value function (v_π) is the unique solution to the Bellman equation for state s [2].

A policy π can be concluded as being better than another policy π' if its expected final cumulative reward $v_\pi(s)$ is greater than or equal to the other policy's expected final cumulative reward $v_{\pi'}(s)$ [2]. The policy with the highest expected final cumulative reward possible is known is the optimal policy, and is represented as π_* [2]. The optimal state-value function is represented by v_* and is formulated as follows [2]:

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

[2]

The optimal action-value function is represented by q_* and is formulated as follows [2]:

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

[2]

The Bellman optimality equation demonstrates that a value of a state under an optimal policy is the highest expected return for the best action from that state [2], and can be formulated as follows:

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

[2]

The Bellman optimality equation for the optimal action-value function can be formulated as such [2]:

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

[2]

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 requires assuming that the Markov property is satisfied [2]. Therefore, approximate solutions are most efficient in terms of computing power and memory available [2].

2.1.2 Dynamic Programming

Dynamic Programming is a set of algorithms that can accurately compute optimal policies under the assumption that a perfect model of the environment's dynamics are provided in the form of an MDP [2]. However, as mentioned earlier, very rarely are perfect environment dynamics available for an environment, but other algorithms attempt to perform close to DP with lesser computational cost and dropping the assumption of a perfect environment [2]. Equations 2.13 and 2.14 can be used in deducing update rules in DP algorithms [2]. The calculation of the value function v_π under policy π is known as policy evaluation or the prediction problem, and the deduction can be seen in equation 2.10 [2]. Assuming the environment's behavior is fully known, the computation of the solutions to the value functions for each state is compute-heavy but achievable, and iterative methods to approximate value functions for a state s are the best approach [2]. The value functions for each state are iteratively updated based on the discounted estimates of the value function for the previous iteration plus the current reward times the probability of each of those values occurring using the Bellman equation for the value function as an update formula as such [2]:

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

[2]

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 if agent traversal termination happens from any state assuming policy π [2]. The complete algorithm for iterative policy evaluation is as follows [2]:

Algorithm 1 Iterative Policy Evaluation to estimate the value function V via v_π

```
  Input: Policy $\pi$ 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 $\Delta < \theta$
```

[2]

Calculating the value function under a policy π is useful in determining policies that increase final cumulative reward [2]. To determine policies that increase final cumulative reward, the policy improvement theorem must be considered, and its general formulation is as follows [2]:

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

[2]

The policy improvement theorem essentially states that if it is better to select an action a from state s based on a policy π' in terms of maximizing the final cumulative reward expected 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 in terms of expected final cumulative reward [2]. The formulation of the policy that leads to policy improvement is the action that maximizes the expected cumulative reward is as follows [2]:

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

[2]

Using v_π, an improved policy π' can be derived, and used to compute and evaluate $v_{\pi'}$ and use it to derive an even more improved policy π'', until an optimal policy is reached in a finite number of iterations due to the MDP that is in focus here being of a finite nature, and this is known as policy iteration [2]. The algorithm for policy iteration is as follows [2]:

Algorithm 2 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
```

[2]

The policy evaluation step in policy iteration is iterative in nature and can be stopped after a single update of each state's value function values instead of waiting for convergence to v_π, and this shorter version of policy evaluation along with policy improvement is known as value iteration [2]. Value iteration essentially merges the value computation step in the policy evaluation and the policy computation step in the policy improvement into one update rule summed over all actions to determine the action that yields the highest expected final cumulative reward. This update step can be formulated as such [2]:

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

[2]

The corresponding value iteration algorithm is as follows:

[2]

Algorithm 3 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 $\Delta < \theta$
```

[2]

Dynamic Programming is not feasible for problems with large state spaces, but compared to other methods for solving MDPs, DP methods are computationally efficient [2]. 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 that the total number of possible deterministic policies is k^n [2]. 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 [2].

2.1.3 Temporal Difference Learning

Temporal Difference (TD) learning merges ideas from Monte Carlo methods with DP methods [2]. 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 [2]. TD methods are similar to DP methods in that they make updates to their approximations of values based on the other learned approximations [2]. TD methods' approach to the prediction problem is what differentiates these methods from DP and Monte Carlo methods, as all of these methods implement some variant of a Generalized Policy Iteration (GPI) to solve the control problem of finding an optimal policy [2]. A TD method known as the $TD(0)$ or one-step TD method waits only till the end of the next time step $t + 1$ and 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 [2]:

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

[2]

This approach differs from Monte Carlo (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 [2]. The $TD(0)$ method's algorithm is as follows:

[2]

Algorithm 4 TD(0) - estimation of the value function V via v_π

```
Input: Policy $\pi$ for evaluation
Parameter: Step size $\alpha \in (0, 1]$
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 $\pi$ for $S$
              Take action $A$, observe $R, S'$
              $V(S) \leftarrow V(S) + \alpha\,[R + \gamma V(S') - V(S)]$
              $S \leftarrow S'$
  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, similar to how DP makes updates to its estimates [2]. The formulation of the estimated v_π as a recursive form of the Bellman equation is as follows [2]:

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

[2]

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 cumulative rewards, whereas DP methods use the current estimates in the form of the last line of equation 2.20 as a target reward [2]. The target return for TD methods uses the estimate of V in the current time step along with the sampled expected values to estimate v_π [2]. 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 [2]:

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

[2]

TD methods do not require a model of the environment's probability distributions of rewards and transition probabilities, and can be used in applications where DP methods cannot [2]. TD methods' estimates are incremented by the next time step instead of at the end of an episode like MC methods, and in applications that have long episodes, this attribute of TD methods proves to be very useful [2]. TD(0) under a constant policy π has been proven to converge to v_π and converge quicker than MC methods holding the step size constant for tasks that are stochastic in nature [2]. TD methods finding estimates of the value function (the parameter in question) that match the maximum-likelihood model of the MDP at every time step (estimating parameters that maximize the probability of observing the training data) while MC methods computing estimates to reduce the mean square error with respect to the training data at the end of each training episode is the reason behind why Batch-training-based TD methods converge quicker than batch-training-based MC methods [2].

There are two types of control problem solutions to finding an optimal policy, on-policy methods and off-policy methods [2]. On-policy methods uses the same policy that explores the environment for the policy improvement while attempting to find an optimal policy. Off-policy methods contain a policy that is used to explore the environment, and a different policy is used for the policy improvement step [2]. 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), and 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 formulation as follows:

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

[2]

The Sarsa algorithm (dependent on the policy's behavior) using an ϵ-greedy policy that chooses an action from the list of possible actions that has the highest action-value function value, but choose a random action with a probability of ϵ is as follows [2]:

Algorithm 5 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

```
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 $\epsilon$-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 ($\epsilon$-greedy policy) deduced from $Q$
              $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$
              $S \leftarrow S'$
              $A \leftarrow A'$
  Until $S$ is terminal
```

[2]

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 value without regard to the policy's behavior from the states that the policy chooses to explore [2]. The Q-learning update rule is formulated as such [2]:

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

[2]

The Q-learning algorithm is as follows [2]:

Algorithm 6 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

```
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 ($\epsilon$-greedy policy) deduced from $Q$
            $Q(S,A) \leftarrow Q(S,A) + \alpha\,[R + \gamma \max_a Q(S',a) - Q(S,A)]$
            $S \leftarrow S'$
    Until $S$ is terminal
```

[2]

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 the PWRLSys 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 PWRLSys is more appropriate to our specific use case.

ArtBot, is a simple supervised and reinforcement learning platform for teaching concepts in ML to primary and secondary education students, with its main purpose being to integrated allowing students to experiment and see ML & RL concepts in action, and digital gamification of the ML 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, and does not go too deep into 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 traditionally via RL, and transfer learning [9]. This platform has Atari 2600 emulator-based games that are converted into a reinforcement learning problem with a score for the game, and whether the game has terminated or not, and provides benchmark results to compare custom agents' performance against [9]. The bench marked results were recorded using a SARSA(λ)

TD control method, which is an extension of Algorithm 5, the SARSA algorithm, from the previous section that 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 that were 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 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 part of graduate-level students' 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, multi-agent systems [10]. Malmo's environment is dynamic, has many objects and agents interacting in it, multiple-step tasks with planning can be required, and the agent's computational resources are limited [10, 18, 19]. Such a testbed can be used to test even more complex RL algorithms than the ones that could be tested with ALE, and can be expected to be part of advanced graduate-level students' curriculum in reinforcement learning.

All of the above described platforms provide either provide lesser amount of detail in terms of learning reinforcement learning concepts, as in the case of ArtBot's simple agent setup and gameplay for secondary school students, or excessive amounts of detail, in the cases of ALE and Project Malmo's complex game details and complex multi-agent interactions respectively. In the next chapter, we propose PWRLSys, which provides the right amount of detail to learn and implement reinforcement learning algorithms for an undergraduate-level course in AI, with an emphasis on students building parts of an RL agent to navigate a puddle world. The focus in PWRLSys is on implementing Q-learning algorithms in an RL agent within a discrete environment, and not on complex multi-agent interactions in continuous environments or model deployment, providing 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 in the puddle world. This system is built using Java [20], Smartfox Server API [21] and Client API in Java [22].

2.3 Python-Java cross-language integration platforms

Before describing the specifics of the Python-Java integration to play the Game of Amazons in the next chapter, it is necessary to review the most relevant platforms for developing a cross-language integration between Java and Python to understand the differences between the various existing platforms, their advantages, and why Py4J is more appropriate to our specific use case.

Java Embedded Python (JEP) is a platform that has Python embedded into Java via the Java Native Interface (JNI) and Python API, and allows for Python code to be developed alongside the Java code, and be accessed via Java through the API of JEP [23] 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 finally communicate the next move to the Java Smartfox Server API for further processing.

Py4J is a platform that allows for Python code to access Java objects dynamically through a JVM gateway [24]. 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, along with providing 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 if they are more comfortable implementing algorithms with it, and sending their computations to Java, which communicates them to the Java-based Smartfox Server API for UBC's COSC 322 course.

Bibliography

[1] Sutton, Richard S. *Reinforcement learning: An introduction.* A Bradford Book, 2018. → pages 5

[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 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

[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] 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

[12] 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.

[13] 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`

[14] 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

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

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

[17] 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

[18] 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

[19] 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

[20] Oracle. *Java SE 23 Documentation.* (n.d.). [Online]. Available: `https://docs.oracle.com/en/java/javase/23/`

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

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

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

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