A Comparative Study of Multi-Agent Reinforcement Learning

on

Real World Problems

by

Sahil Badyal

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2021 by the
Graduate Supervisory Committee:

Stephanie Gil, Chair
Dimitri P. Bertsekas
Yingzhen Yang

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

This work investigates the multi-agent reinforcement learning methods that have applicability to real-world scenarios including stochastic, partially observable, and infinite horizon problems. These problems are hard due to large state and control spaces and may require some form of intelligent multi-agent behavior to achieve the target objective. The study also introduces novel rollout-based methods that provide reasonable guarantees to cost improvements and obtaining a sub-optimal solution to such problems while being amenable to distributed computation and hence a faster runtime. These methods, first introduced and developed for single-agent scenarios, are gradually extended to the multi-agent variants. They have been named multi-agent rollout methods. The problems studied in this work target one or more aspects of three major challenges of real-world problems. Spider and Fly problem deals with stochastic environments, multi-robot repair problem is an example of a partial observation Markov decision problem or POMDP, whereas the Flatland challenge is an RL benchmark that aims to solve the vehicle rescheduling problem. The study also includes comparisons to some existing methods that are used widely for such problems as POMCP, DESPOT, and MADDPG. The work also delineates and compares different behaviors arising out of our methods to other existing methods thereby positing the efficacy of our rollout-based methods in solving real-world multi-agent reinforcement learning problems. Additionally, the source code and problem environments have been released for the community to further the research in this field. The source code and the related research can be found on https://sahilbadyal.com/marl.

ACKNOWLEDGMENTS

First and foremost I would like to express my gratitude to my advisor and mentor Dr. Stephanie Gil for her unwavering support and guidance throughout these years. Her presence made it possible for me to push my limits, strive for excellence, and work towards the completion of this work.

I would like to thank Dr. Dimitri Bertsekas for being a major inspiration for this work. His work and ideas in this field have paved a way for this research. His energy and zeal towards this work are contagious. Throughout the course of this work, we have had numerous discussions and his feedback and insights have been immensely helpful for the advancement of this research. I would also like to thank Dr. Yingzhen Yang for his support during the past year. I feel incredibly fortunate to have been associated with great mentors.

I would also like to thank my colleagues from the REACT lab, particularly Sushmita Bhattacharya, Siva Kailas, Calvin Norman, and others who have spent their time discussing and working on the ideas presented in this work. I would also like to thank Arnav Dhiman, Akarshit Wal and Ishita Thakur for their useful feedback on the chapters.

Finally, I want to thank my family, especially my parents for their incredible love and support. This work is dedicated to my grandparents who believed in me from my very childhood.

Thank you.

Sahil Badyal

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Symbol	Definition
s	Denotes the system state.
S	Denotes the set of all possible states.
k	Denotes the stage or step.
K	Denotes the length of horizon for finite horizon problems.
x_k	Denotes the system state at stage k.
u_k	Denotes the control or action at stage k.
U	Denotes the set of all possible controls.
f	Denotes the state transition function.
w	Denotes the random variable that models the system disturbance.
p	Denotes the transition probability in a MDP.
z_k	Denotes the current observation in POMDP.
Z	Denotes the set of all possible observations.
b_k	Denotes the belief state at stage k.
y_k	Denotes the feature state at stage k.
g_k	Denotes the cost at stage k.
α	Denotes the discount factor.
γ	Denotes the learning rate in neural netowork approximations.
J	Denotes the cost function.
m	Denotes the number of agents.
M	Denotes the set of all agents.
Q	Denotes the Q-factor.
$\tilde{Q}$	Denotes the approximate Q-factor.
μ	Denotes the static policy.
$\tilde{\mu}$	Denotes the approximate policy.
μ^*	Denotes the optimal policy.
π	Denotes the set of policies for an episode.
Π	Denotes the set of all possible policies.
$\tilde{J}$	Denotes the approximate cost function.
J^*	Denotes the optimal cost function.
T	Denotes the Bellman Operator.
$\mathbb{E}$	Denotes the expectation.

Chapter 1

INTRODUCTION

Reinforcement Learning (RL) is one of the major paradigms of machine learning and artificial intelligence. It aims at the development of methods that can learn behaviors to optimize the performance of an agent (e.g. a robot) at a task, by maximizing a reward signal (or minimizing a cost signal) received from the environment. This form of goal-driven learning paradigm is quite natural to Humans as it is based on the interaction of an agent with its environment depending upon the stimuli received. This is unlike other learning paradigms like supervised and unsupervised learning, where learning is dependent on the collected data. These behaviors, also known as policies, are defined by the controls or actions of the agent at various system states of the environment. Most of the ideas in reinforcement learning have been compiled in books [34],[12], and [35].

Another interesting aspect of RL is observed in the problems that require multiple agents either due to the scale, complexity, or inherent nature (amenability) of the environment itself. Examples could be forest fire/disaster management, search and rescue, mechanical repair of gas pipelines, etc. All these applications require multiple robots exhibiting some form of intelligent behaviors, for instance, surrounding a target object in search and rescue problems, divide and conquer to manage the aftermath of a nuclear disaster, etc., depending upon the state of the environment and objective of the task. This sub-field of reinforcement learning which deals with the design of algorithms that use multiple agents for solving an objective is termed as Multi-Agent Reinforcement Learning (MARL). There are additional challenges to such problems like large control and state space, communication, partial observation, stochastic environments, etc. This work presents the methods specifically designed to solve some of these challenges.

In **Chapter 1** we present the general background and challenges of RL with a special focus on MARL. **Chapter 2** discusses the key ideas (using consistent mathematical notations) in reinforcement learning which are essential in understanding the methods discussed in the work. **Chapter 3** introduces the our work on rollout based methods, their taxonomy. and the key ideas that led to their development. **Chapter 4** presents the three real-world problems, their detailed definition and the corresponding experimental results on our multi-agent rollout variants. We also prensent a comparison to the other existing methods. We conclude our discussion and posit a future direction in **Chapter 5**.

1.1 Basic Elements of Reinforcement Learning

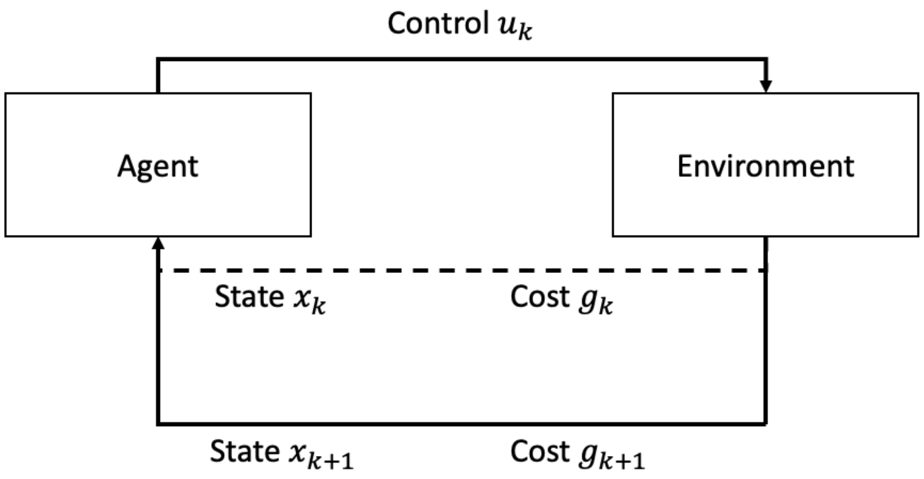


Figure 1.1: Reinforcement Learning Overview.

There are primarily two basic elements of Reinforcement Learning namely *agent,* and an *environment.* The entire domain of RL works on the interaction between these two elements as we show in figure 1.1. These basic elements further contain some sub-elements and together they provide a general framework for all reinforcement learning problems and hence it is important to discuss them here.

Environment is defined by a system that contains the basic building blocks of the problem like a state (current situation), an interface for the agent to interact, the manifestation of the task itself, ability to change its state depending upon the external interactions or internal process. Additionally, it also refers to the world in which our agent resides and defines the boundaries or limits to the execution of actions. Mathematically, it is defined as a set of states that are affected both by some internal known/unknown process and the interactions with the agent. The objective of an agent is to use its controls or actions to reach a desired goal state in the environment.

Agent refers to the actor responsible for solving the problem or achieving the desired objective. It might be a computer, an actuator, a robot, or a virtual game player. An agent is a complex entity that has four important sub-elements namely *policy, cost/reward, cost/value function*, and optionally a *model.* We now discuss them in detail.

Policy refers to the mapping of a state to action or control and this defines the behavior of the agent. Although it is important to note here that it is not necessary for the agent to know the actual state of the environment, for example in partially observable Markov decision process (POMDP), the agents just have a belief state which is a probabilistic estimation of the actual environment state. The agent estimates the state (perfect or probabilistic) of the environment as observation through the sensors and then uses the policy learned through the RL algorithm to reach the goal. It is sufficient to say that the goal of any RL method is to learn an optimal policy.

Cost/Reward is an important signal required to estimate the quality of the learned policy. Generally, every state transition yields an associated reward or cost (except the final state that has a terminal cost) that is determined by the agent based on some pre-defined metric of the problem statement. For instance, in a search and rescue scenario, the reward could be defined by the proximity to the lost person, whereas in pipeline repair the cost could be defined by the number and the disrepair of damaged sites. It is also important to note that reward and cost are essentially the same signals with their sign reversed i.e $cost = -reward$. For the sake of simplicity, we will now use cost during the scope of this work. Designing the cost signal, i.e. defining the cost of a state transition is of fundamental importance in formulating an RL problem.

Cost/Value Function defines the overall quality of the state by taking into account the future course of states that can be reached from the state through the policy. A state might have a high immediate cost but a low value of cost function suggesting better suitability. The cost (also known as stage cost) helps to gauge the immediate desirability of a state, whereas the cost function is used to determine the long-term desirability. Another noteworthy difference is that cost is a part of RL problem formulation whereas cost function is dependent on the algorithm used to estimate it. It would be an understatement to state that the quality of an RL method depends on the quality of its cost function estimation.

Model is an important but optional sub-element used for planning, that helps us to predict the state transition of an environment in simulation. While there are some problems where it may not be required, as the agent can learn through multiple trials in the real environment, but in most practical applications a model is desired. Most RL problems can be formulated as Markov Decision Processes (MDP), which will be introduced and discussed in detail in the next chapter.

The agent enters the system and observes the so called *initial state*, then uses a policy to reach the *desired goal state* in the environment. Once the desired state is reached we terminate the execution of the agent and call this an **episode**.

1.2 Multi-Agent Reinforcement Learning

The multi-agent reinforcement learning framework is almost similar to the single-agent counterpart except that the state and consequentially the cost function is not only dependent on the policy of a single agent rather the policies of a set of agents. Another way to formulate it is that the control component can be broken down to m components, where m is the number of agents. Thus the actual state transition and cost of the environment is dependent on the join control space that is exponential to the number of agents. Here an agent has to optimize its cost function which is now dependent on the policies of other agents. Additionally, some interesting scenarios exist in MARL based on the type of tasks and agent behavior.

Co-operative scenario is probably the simplest and most common setting, where the agents have a common shared cost and goal. Therefore the agents try to optimize their local policies, in a way that brings them a step closer to the objective. The agents can have a local (private) cost which can then be averaged at the global level, thus enabling a distributed setup. This forms the basis of the most common sequential multi-agent MDP. Examples include search and rescue problems, pipeline repair problems, etc.

Competitive scenario arises when there are multiple agents with conflicting goals. This commonly arises in a lot of computer games and has been studied extensively as two-player games. Such scenarios are typically modeled as *zero-sum Markov games*. Here, one agent tries to maximize the cost while the other tries to do the opposite. The optimum performance is reached at a reward of zero. This can be used to develop robust controllers that can handle adversarial attacks.

Mixed scenario can arise out of the various combinations of both cooperative and competitive agents. It can also arise in the cases when each agent has its own goal, that may or may not be conflicting with others and is modeled as a *general-sum game*.

1.3 General Challenges

We can summarize the challenges in reinforcement learning in two parts, the ones common to almost all real-world RL problems, and the ones arising because of multi-agent scenarios.

1.3.1 RL Challenges

The general RL challenges in real-world problems have been discussed in depth in [18]. We present a summary of these challenges here.

1. **Large State Space**

   One of the major challenges in reinforcement learning is the size and dimensionality of state space (possible situations). Any optimization algorithm operating in a high dimensional state space suffers from the *curse of dimensionality* i.e. the exponential rise of the number of states with an increase in state variables. This means that the exploration of state space becomes challenging. The balance between *exploration* and *exploitation* is a central theme to almost all reinforcement learning problems. We need our algorithm to explore the state space to find favorable states but also to remember and exploit the already explored states with a low value of cost function. Higher-dimensional state-space poses a big challenge in exploration and this motivates the use of approximate methods that target sub-optimal solutions to the problems.

2. **Environment Modeling**

   Another challenge in practical reinforcement learning is to model the real environment. Modeling a real-world process may be difficult either because of the complexity of the process or the high variability in the process itself might render its modeling impractical. Although not necessary in many cases, in general, a model is required for multiple reasons including risky or harmful environments like nuclear sites, non-reproducible environments like the aftermath of disasters, resetting problems (to reset the experiment again for the next episode), to name a few. But developing an accurate model of any real-world process requires a lot of data and analysis, which is often not sufficient to yield a reliable model for the planning phase of RL algorithms.

3. **Re-Planning**

   Re-planning is a crucial aspect of a robust RL algorithm. Based on the new information presented to the agent, it should be able to re-plan and adjust its policy to reach the desired goal. But this is not a trivial task as it often requires re-computations, which are costly and time-consuming. Re-planning is also required in highly stochastic environments especially when the state space is large and the environment is difficult to model. One of the very recent and popular problems in the RL community is the Flatland Challenge [26], which has been discussed in this work as it addresses the vehicle rescheduling problem [22] (VRSP), which is essentially a re-planning problem.

4. **Partial Observation and Stochasticity**

   Another issue with most real-world problems is partial observability. The agent might not be able to observe the entire state of the environment due to sensory or environmental limitations and in such cases has a probability distribution (or a belief) over the possible states of the system. It is also possible for the system to be stochastic, for instance, in the case of search and rescue the position of the missing person might be stochastic due to aimless wandering. Another big challenge associated with POMDP problems is the *curse of history*, which will be discussed in chapter 2. Partial observation may also be related to non-stationarity which is discussed in the next subsection. These challenges have been explored in-depth in this work. The pipeline repair, and spider and fly problems are examples of such systems.

1.3.2 MARL Challenges

The multi-agent nature of the problem, beings about some special challenges which have been discussed in [39] and [18]. Here are some of the major challenges in the field.

1. **Large Control Space**

   One of the immediate repercussions of deploying any RL algorithm on multiple agents is the exponential explosion of control space (possible controls). At every time step, each agent is free to take a valid control out of its control set and thus the joint control space grows exponentially to the number of agents. This causes most MARL algorithms to be bad at scaling to a higher number of agents. This calls for the development of sophisticated algorithms that can address this problem. We develop such algorithms as a part of this work which is discussed in chapter 3.

2. **Multi-Dimensional Learning Goals**

Unlike minimizing the long-term cost, efficiently as in single-agent RL, the MARL agents might have objectives that are either multi-dimensional, conflicting, or at times vaguely defined. It is difficult to understand and even define the central objective of agents in a MARL problem.

3. **Non-stationarity**

   One problem that is quite commonplace in MARL is the notion of a non-stationary environment due to the concurrent learning of policies by the agents. To an agent, the environment might seem stochastic as every agent's control impacts the observation of others. This might cause the algorithm to produce policies that are oscillatory and never converge to an optimal/sub-optimal solution.

4. **Communication Constraints**

   Since the agents should optimize the given task in a joint control space, they need fast and reliable communication when deployed in the real world. This is a crucial yet relatively less explored side of MARL algorithms that has been addressed in a few of our methods discussed in chapter 3.

1.4 Summary

This chapter provides a brief theoretical overview of the basics of RL with a special focus on the MARL. Our objective was to get familiar with the terminology and thus fix a common vocabulary on which the entire work is based so that we can progress towards a more formal definition of these concepts. From the next chapter, we will introduce the key ideas behind reinforcement learning that are necessary to understand the work.

Chapter 2

BACKGROUND

In this chapter we provide some background on the various methods and ideas in reinforcement learning and discuss some of them in detail. We do so using the mathematical notation that would be introduced and then reused consistently throughout the course of this work. We begin by first formulating the model of the environment and pose it as a Markov Decision Process (MDP). We discuss briefly the partial observation case of MDP, also known as POMDP. We then delve into the basics of Dynamic Programming (DP) algorithm, moving quickly to value and policy iteration algorithms. We then introduce a key method that would form the base of this work i.e. *Rollout*. Next we talk about various approximations to solve an MDP to get sub-optimal solutions. We also discuss the Monte-Carlo, Policy Gradient and TD methods, that have been very effective in solving some POMDP problems and have been used as a comparison to our novel work. We end by discussing MARL, defining the basic notations, theory, and some existing solutions like MADDPG and QMIX. These topics have been carefully chosen as they are fundamental to the work presented here. Fig. 2.1 presents the general taxonomy of various types of RL methods.

2.1 Markov Decision Processes

Most environments in RL literature can be formulated as a Markov Decision Processes (MDP) i.e. the processes that follow *Markov assumption*. An MDP (shown in figure 2.2) consists of a set of N [1] states $S = \{s_0, s_1, ..., s_N\}$, with a well-defined probability of transition $p_{ij}(u)$ (i denotes state s_i and j denotes state s_j) between the

[1]It is possible for N to be infinity, making it an infinite state MDP

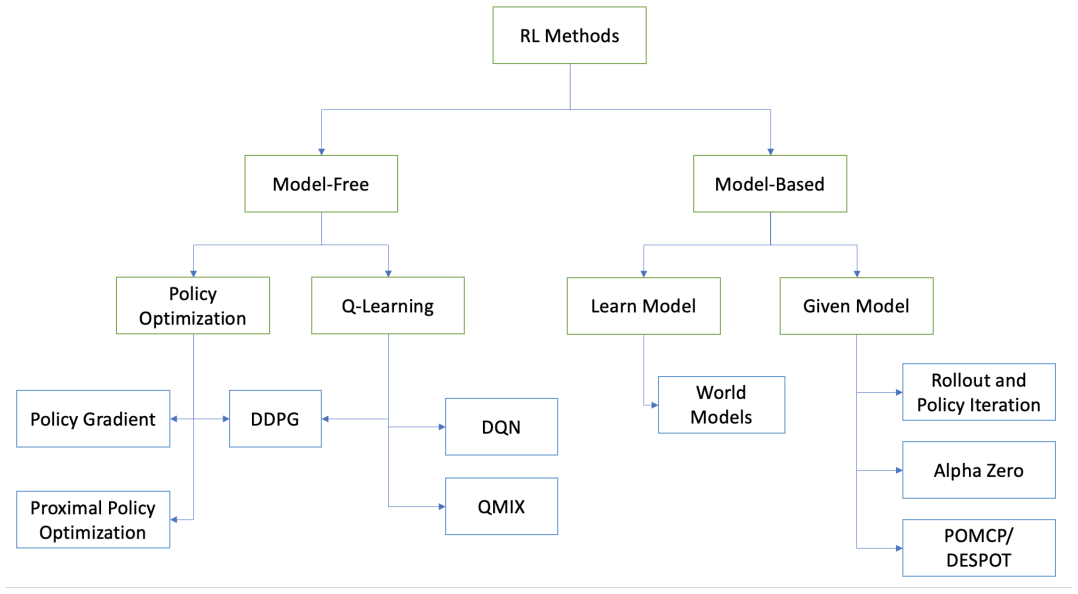


Figure 2.1: A Non-exhaustive, but Useful Taxonomy of Modern RL Methods Adopted from [1].

states. The agents interact with the environment at discrete time-steps (also known as stages) denoted by $k = 1, 2, 3, ..., K^2$, where K is the length of horizon. At each step, the agents observe the environment and get the snapshot of their current state ${x_k}^3 \in S$. The agent can then choose an optimizing control $u_k \in U_k(x_k)$ and incur a cost $g_k(x_k, u_k, w_k) \in G$ while transitioning to a new state x_{k+1} using a transition probability $p_{ij}(u_k)$ and disturbance w_k. Here w_k is a random variable characterized by the distribution $P(...|x_k, u_k)$. Figure 2.3 clearly depicts this definition. Mathematically, state transition can also be written as:

$$x_{k+1} = f_k(x_k, u_k, w_k) \tag{2.1}$$

[2]K determines the length of horizon of the algorithm i.e. time steps required to reach the goal state. For some problems this might be infinite and they are termed as infinite horizon problems.

[3]Note that x_k here denotes the state reached at time step k, it takes one of the states in set S. For instance, it is possible for x_k to be equal to x_{k+1}.

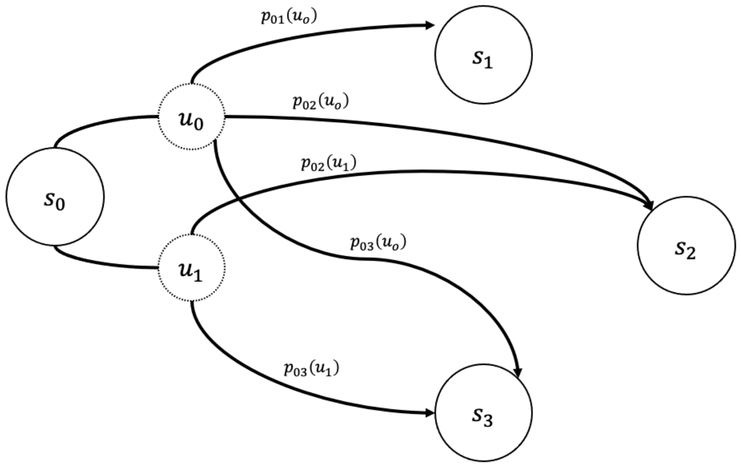


Figure 2.2: An MDP with State Transition from Initial State s_0.

Under *Markov assumption*, the current state encompasses all the necessary information for the agent to make a decision, and thus the history of its interaction with the environment does not matter. Mathematically,

$$P(x_{k+1}|x_0, u_0..., x_k, u_k) = P(x_{k+1}|x_k, u_k)$$

For a finite MDP [4] , the sets S, U and G are finite sets, and the random variables g_k, x_k have a well defined discrete probability distribution.

We also define the additive cost $J(x_0, u_0, ..., u_K)$ of the entire trajectory starting from the initial state x_0 and under the control sequence $\{u_0, ..., u_K\}$ as

$$J(x_0, u_0, ..., u_K) = g_K(x_K) + \sum_{k=0}^{K-1} g_k(x_k, u_k, w_k) \tag{2.2}$$

To get an optimal cost we must minimize this cost J over the set of all possible controls. But in practice the optimization is not performed on control sequence rather a sequence of functions or *policies*, $\pi = \{\mu_0(x_0), ..., \mu_K(x_K)\}$, where $\pi \in \Pi$. As

[4]MDP is generally assumed to be stochastic but the deterministic version is just a special case with no noise or disturbance w_k.

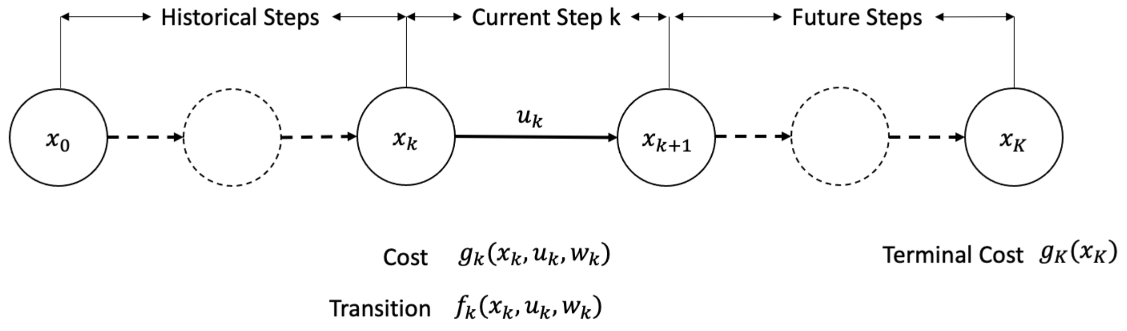


Figure 2.3: State Transition in a Finite State, Finite Horizon MDP.

discussed in chapter 1, the policy is a function that maps a state to a control. This formulation is more general and hence eq. (2.1) can be re-written as:

$$x_{k+1} = f_k(x_k, \mu_k(x_k), w_k) \tag{2.3}$$

Additionally, the cost function now is not just a sum, but an expectation of the cost through the stochastic trajectories. Consequently eq. (2.2) transforms to:

$$J_\pi(x_0) = \mathop{\mathbb{E}}_{w_k}\left\{ g_K(x_K) + \sum_{k=0}^{K-1} g_k(x_k, \mu_k(x_k), w_k) \right\} \tag{2.4}$$

$$J^*(x_0) = \min_{\pi \in \Pi} J_\pi(x_0) \tag{2.5}$$

Note here that we are now minimizing over a sequence of policies and not just a single policy or controls, and this significantly increases the complexity of search space. This propels the need for using Monte Carlo Simulation and approximate methods to solve such problems. Now we increase the complexity of MDP by introducing a more realistic and challenging case of partial observation.

2.1.1 Infinite Horizon Problems

Now we discuss problems that do not take a finite number of steps to reach the goal state. In such problems either it is not possible to reach the absolute goal state and hence the agent keeps on trying to be as close as possible to the target state, one example could be a continuous state problem where the goal is to survive in the environment for as long as possible. Other problems could be where the process of the system causes the states to transition at each step thereby making the goal non-stationary and hence the optimization process might go on infinitely. An example of such a process is a pipeline problem, in which even after fixing the pipeline, it might fall into disrepair after a few steps due to subsequent wear and tear. Since K is infinite, we aim to minimize the total cost over infinite step using:

$$J_\pi(x_0) = \lim_{K \longrightarrow \infty} \mathop{\mathbb{E}}_{w_k} \left\{ \sum_{k=0}^{K-1} \alpha^k g(x_k, \mu_k(x_k), w_k) \right\} \tag{2.6}$$

where α is the discount factor which lies in the interval $(0, 1]$. Alpha is known as discount factor, as for $\alpha < 1$ as it discounts the cost of the future states and puts more emphasis on the cost of nearby states. Such settings are quite common in a lot of real-world problems. We will elaborate more about such problems in future sections.

2.1.2 Partial Observation

In the previous section, we have assumed that the agents can perfectly sense the environment and form an accurate estimate of the state. But as we know from real-world examples this is rarely the case. There are always some variables that are hidden from sensors that may affect the state, alternatively, the limitations of sensors might make it impossible to deduce the entire state of the system with certainty.

Such processes are what we call the *Partially Observable Markov Decision Processes* (POMDP). Since the agent cannot fully observe the system, it observes a partial information/observation $z_k \in Z$ with probability $p(z_k|s_i, u_k)$. The objective of the agent is to optimize overall controls given the observation history. Here history refers to the sequence of observations z_k and controls u_k i.e. $\{z_0, u_o, ..., z_k, u_k, ...\}$ and as the observation history can be large (for instance infinite horizon problems), this causes another significant challenge in POMDPs called *the curse of history.*

Mathematically speaking a POMDP introduces the notion of belief state b_k which is a probabilistic interpretation of the actual state of the environment. In particular the belief state is a probability vector $b_k = \{b_0(s_0), ..., b_{N-1}(s_{N-1})\}$, where $b_k(s_i)$ is the probability of the current state being s_i. Here it is not difficult to see that a POMDP becomes intractable in an infinite or continuous state scenario. We will talk about POMDP in detail in the next chapter, where we introduce our methodology.

2.2 Dynamic Programming (DP)

Since its introduction in the early 1950's *dynamic programming* has become an important optimization technique and also a computer programming method. It is used widely in a variety of problems ranging from simple and tractable optimizations to complex and intractable ones. Consequently, it forms the basis for most RL optimization methods. The algorithm is based on a simple idea of *the principle of optimality.*

Principle of Optimality simply states that *the tail of an optimal sequence should be an optimal sequence for tail sub-problem.* For RL this translates to, that given an optimal control sequence $\{u_0^*, ..., u_{K-1}^*\}$ and corresponding optimal state sequence $\{x_0^*, ..., x_{K-1}^*\}$ if we consider a sub-problem that begins at time k, whereby the optimal cost to go from step k (tail sub-problem from k) to $K-1$ given by

$$\min_{u_j \in U_j(x_j)} \left\{ g_k(x_k^*, u_k) + \sum_{j=k+1}^{K-1} g_j(x_j, u_j) + g_K(x_K) \right\}$$

then the sub-sequence $\{u_k^*, ..., u_{K-1}^*\}$ is an optimal sequence for this tail sub-problem from k.

For a stochastic finite horizon MDP, the dynamic programming method is the following:

Begin with:

$$J_K^*(x_K) = g_K(x_K), \qquad \forall x_K$$

If we assume that:

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \mathop{\mathbb{E}}_{w_k} \left\{ g_k(x_k, u_k, w_k) + J_{k+1}^*(f_k(x_k, u_k, w_k)) \right\}, \qquad \forall x_k \tag{2.7}$$

and $u_k^* = \mu_k^*(x_k)$ minimizes this equation, then the set of optimal policy is given by $\pi^* = \{\mu_0^*, ..., \mu_{K-1}^*\}$.

This optimal control sequence $\pi*$ is calculated an a minimal argument that generated this optimal cost. To construct the optimal sequence:

Begin with:

$$\mu_0^*(x_0) = \arg \min_{\mu_0(x_0) \in U_0(x_0)} \mathop{\mathbb{E}}_{w_0} \left\{ g_0(x_0, \mu_0(x_0), w_0) + J_0^*(f_0(x_0, \mu_0(x_0), w_0) \right\}$$

and

$$x_1^* = f_0(x_0, \mu_0^*(x_0), w_0)$$

Next go forward, for $k = 1, ..., K-1$, set

$$\mu_k^*(x_k) = \arg \min_{\mu_k(x_k)\in U_k(x_k^*)} \mathop{\mathbb{E}}_{w_k} \left\{ g_k(x_k^*, \mu_k(x_k), w_k) + J_k^*(f_k(x_k^*, \mu_k(x_k), w_k) \right\}$$

and

$$x_{k+1}^* = f_k(x_k^*, \mu_k^*(x_k), w_k)$$

Q-Factor Minimization

Q-Factor minimization also known as action-value minimization is an equivalent form of the DP method, the only difference being that instead of minimizing the cost function J_k^*, we generate *Q-factors* for all combination of state-control pairs.

Mathematically, at step k

$$Q_k^*(x_k, \mu_k(x_k), w_k) = \mathop{\mathbb{E}}_{w_k} \left\{ g_k(x_k, \mu_k(x_k), w_k) + J_{k+1}^*(f_k(x_k, \mu_k(x_k), w_k)) \right\} \tag{2.8}$$

With this definition the equation 2.7 for optimal cost function at step k can be re-written as:

$$J_k^*(x_k) = \min_{\mu_k(x_k)\in U_k(x_k)} Q_k^*(x_k, \mu_k(x_k), w_k) \tag{2.9}$$

One of the benefits of this reformulation is that we can get rid of the cost function J completely and write the recursive definition of Q-factors as:

$$\begin{aligned} Q_k^*(x_k, \mu_k(x_k), w_k) = \mathop{\mathbb{E}}_{w_k} \Bigg\{ & g_k(x_k, \mu_k(x_k), w_k) + \\ & \min_{\mu_{k+1}(x_{k+1})\in U_k(x_{k+1})} Q_{k+1}^*(x_{k+1}, \mu_k(x_{k+1}), w_{k+1}) \Bigg\} \end{aligned} \tag{2.10}$$

The various forms of methodologies that arise out of this formulation are collectively termed as *Q-Learning* and will be discussed later. One immediate benefit of this method as you might see is that Q-factor minimization works on the expectation of minimization over future Q-factors rather than minimization of multiple expectations. This is particularly useful in online methods in a model-free setting as Q-factor can be estimated through multiple runs in the environment.

2.3 Value Iteration (VI)

Value iteration algorithm results from the DP formulation, that uses the DP eq. (2.7) for iterative estimation of the value of each state. N iterations of such minimization over these values yield the optimal control sequence. The VI algorithm is based on the *Bellman's Equation* (shown below) that holds for each state $s \in S$.

$$J^*(s) = \min_{u \in U(s)} \mathop{\mathbb{E}}_{w} \left\{ g(s,u,w) + \alpha J^*(f(s,u,w)) \right\} \tag{2.11}$$

alternatively in terms of *Q-factors* as

$$J^*(s) = \min_{u \in U(s)} Q(s,u) \tag{2.12}$$

Many a times this equation is also written using a special symbol T, which is called the *Bellman Operator* as follows:

$$J^*(s) = (TJ^*)(s) \quad \forall s \in S \tag{2.13}$$

We can thus, write the VI algorithm as:

Algorithm 1: Value Iteration Algorithm

```
1  Initialize the value of a state $J(s)$ for all states $s \in S$ and define a small
     threshold $\theta > 0$ for determining the accuracy of value estimation.
2  $\delta = +\infty$
3  while $\delta >= \theta$ do
4      $\delta := 0$
5      forall $s \in S$ do
6          $J_{old}(s) := J(s)$
7          $Q(s,u) := \mathbb{E}_w \left\{ g(s,u,w) + \alpha J(f(s,u,w)) \right\}$
8          $J(s) := \min_{u \in U(s)} Q(s,u)$
9          $\mu(s) := \arg\min_{u \in U(s)} Q(s,u)$
10         $\delta := \max(\delta, \|J_{old}(s) - J(s)\|)$
11     end
12 end
```

It is easy to see that this algorithm scales with the number of states of a system, which is exponential in system variables. So it is not surprising that VI is unsuitable for problems with very large state spaces. It is also important to note that this equation is not dependent on the horizon K and for finite-horizon problems, we can set α to 1.

2.4 Policy Iteration (PI)

In this section, we introduce the idea of *Policy Iteration* (PI), an effective and alternative method of reinforcement learning that works on the principles of *policy evaluation* and *policy improvement*. We start with a random stationary policy μ^0 and generate a sequence of improved policies $\mu^1, \mu^2, ...$ by the iterative application of policy evaluation and improvement. Under the exact scenario, it has convergence guarantees to optimal policy μ^*.

Policy Evaluation deals with evaluating the cost of each state by the application of the *Bellman's Equation* using a stationary policy. For a policy μ^i (i.e. the policy obtained at i^{th} policy iteration) this can be written as:

$$J_{\mu^i}(s) = \mathop{\mathbb{E}}_{w}\left\{ g(s, \mu^i(s), w) + \alpha J_{\mu^i}(f(s, \mu^i(s), w)) \right\} \tag{2.14}$$

Like the VI algorithm, to find $J_{\mu^i}(s)$, we use multiple iterations to converge to the final value of the policy μ^i.

Algorithm 2: Policy Evaluation Algorithm

```
1  For policy, $\mu^i$, initialize the value of a state $J(s)$ for all states $s \in S$ and a
     small threshold $\theta > 0$ for determining the accuracy of value estimation.
2  $\delta = +\infty$
3  **while** $\delta >= \theta$ **do**
4      $\delta := 0$
5      **forall** $s \in S$ **do**
6          $J_{old}(s) := J(s)$
7          $J(s) := \mathbb{E}_w \left\{ g(s, \mu^i(s), w) + \alpha J(f(s, \mu^i(s), w)) \right\}$
8          $\delta := \max(\delta, \|J_{old}(s) - J(s)\|)$
9      **end**
10 **end**
```

Policy Improvement follows a minimization over all controls on the eq. (2.14) to find a better and improved policy. Hence we can write:

$$\mu^{i+1}(s) = \arg \min_{u \in U(s)} \mathop{\mathbb{E}}_{w}\left\{ g(s, u, w) + \alpha J_{\mu^i}(f(s, u, w)) \right\} \tag{2.15}$$

These two processes are repeated in an alternating iterative fashion (see Fig. 2.4) till convergence i.e. till $J_{\mu^{i+1}} = J_{\mu^i}$

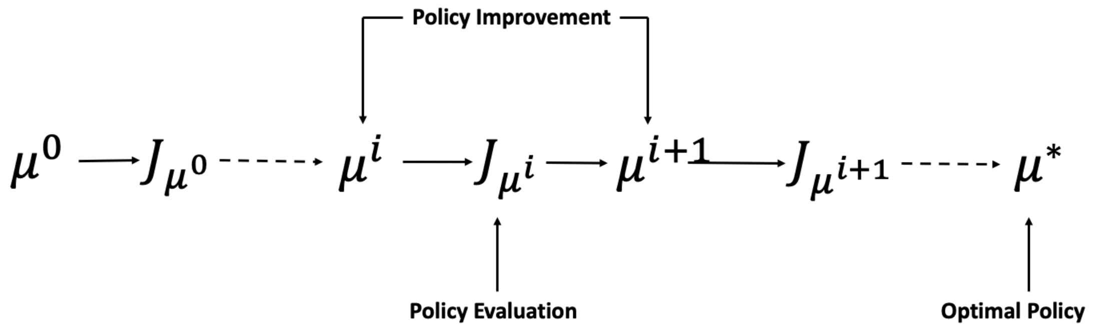

Policy Improvement
$\mu^0 \longrightarrow J_{\mu^0} \dashrightarrow \mu^i \longrightarrow J_{\mu^i} \longrightarrow \mu^{i+1} \longrightarrow J_{\mu^{i+1}} \dashrightarrow \mu^*$
Policy Evaluation
Optimal Policy

Figure 2.4: Policy Iteration Starting from Initial Policy μ_0.

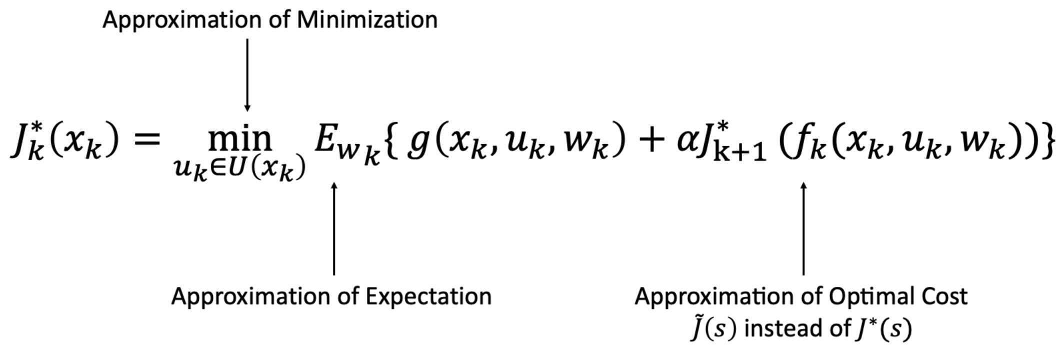
2.5 Approximations

We now revisit the standard DP equation, but posit that its direct application is intractable for most real-world stochastic and POMDP like problems. So it is imperative to look for some approximate and sub-optimal solutions for solving such optimizations. There are two major types of such approximations namely value space and policy space approximations [5] . We will now discuss them one by one.

2.5.1 Approximation in Value Space

Looking at the DP equation (2.11) it is quite intuitive that the intractability arises out of one or more of the three major operations in the equations. As shown in Fig. 2.5, we can approximate the three operations of minimization, expectation, and the

[5]There is also another type of approximations like problem space and feature space but omit their discussion for the scope of this work.


Approximation of Minimization
Approximation of Expectation
Approximation of Optimal Cost
$\tilde{J}(s)$ instead of $J^*(s)$

$$J_k^*(x_k) = \min_{u_k \in U(x_k)} E_{w_k}\{\, g(x_k, u_k, w_k) + \alpha J_{k+1}^* \,(f_k(x_k, u_k, w_k))\}$$

Figure 2.5: The Three Approximations in Value Space.

cost function J^*.

Approximation of minimization is suitable for problems with large control spaces, where it is not feasible to look into all the controls. Such type of approximation has been done in a few methods [38], which will be discussed later.

Approximation of expectation arises quite frequently in most simulation-based algorithms including the ones developed in this work. In real-world environments, the disturbance w_k could be continuous and hence the expectation could be carried out by *Monte-Carlo* methods. We would discuss such methods in detail in section 2.6.

Approximation of optimal cost also known as *approximate cost function* denoted by $\tilde{J}$ is one of the most widely used approximation methodologies. One way to approximate the cost function is using a neural network, which is known to be universal approximator [16],[19]. It is appropriate to say that most of the methods in reinforcement learning (including ours) use some form of cost function approximation. One additional benefit of using $\tilde{J}$ instead of J^* is that we just need a single approximator $\tilde{J}$ instead of $\tilde{J}_1, ..., \tilde{J}_k, ..., \tilde{J}_K$ for a K step horizon problem. This is particularly important for infinite horizon cases. Mathematically, we can write

$$\tilde{J} = g(..., \theta)$$

where g is the approximator function and θ is a set of parameters governing the output of the function. If we use a neural network as the approximator then θ would be the weights that can be updated using back-propagation by minimizing the root mean square error (RMSE) loss between actual optimal cost and the one predicted by the approximator.

Approximation in q-factor space is also a type of value space approximation, but instead of the cost of a state, we approximate the action-value or q-factor $\tilde{Q}$. This might be computationally desirable in a lot of cases, especially because we get away with the expectation in the later stages. In order words, we combine the expectation and value approximation in one step. This forms the basis of one of the major RL methodology called *Q-learning*, which will be discussed in section 2.7.1.

2.5.2 Approximation in Policy Space

Another type of approximation quite common in the infinite horizon cases is the *approximation in policy space.* Here we use a policy from a class of parameterized policies (which may be restricted). For instance, we can introduce a class of policies parameterized by variable θ_k

$$\hat{\mu}_k(x_k, \theta_k) \qquad \forall k \in 0, ..., K-1$$

We can estimate or learn the variable θ_k through some learning processes similar to the value approximator. In a finite control space, this approximator can be a neural network classifier that classifies the state into one of the controls based on the parameters learned through the application of back-propagation (through gradient descent like methods) by minimizing the miss-classification rate (cross-entropy loss). To put it simply, we learn the optimal mapping between state and optimal control in an approximate setting using a machine learning classifier.

One benefit of this approximation over the value space approximation is that if we learn the policy, we can omit the minimization while running the approximate implementation of the policy. This makes the policy space approximation suitable for running the learned sub-optimal policy, where we would want to avoid online costly minimization (cases of stationary optimal policies). Fig. 2.6 shows that the policy space approximation can be viewed as a controller that has been learned using some machine learning technique.

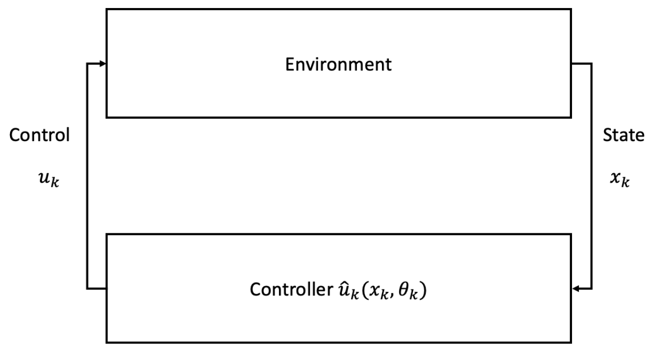


Figure 2.6: Approximation in Policy Space Emulates a Controller.

2.5.3 Rollout and Policy Improvement

In this section, we try to tie down both approximation in value and policy space into an ingenious methodology. We postulate that both of these methodologies can be used to derive the other. This alternating process is the main idea behind *rollout*,

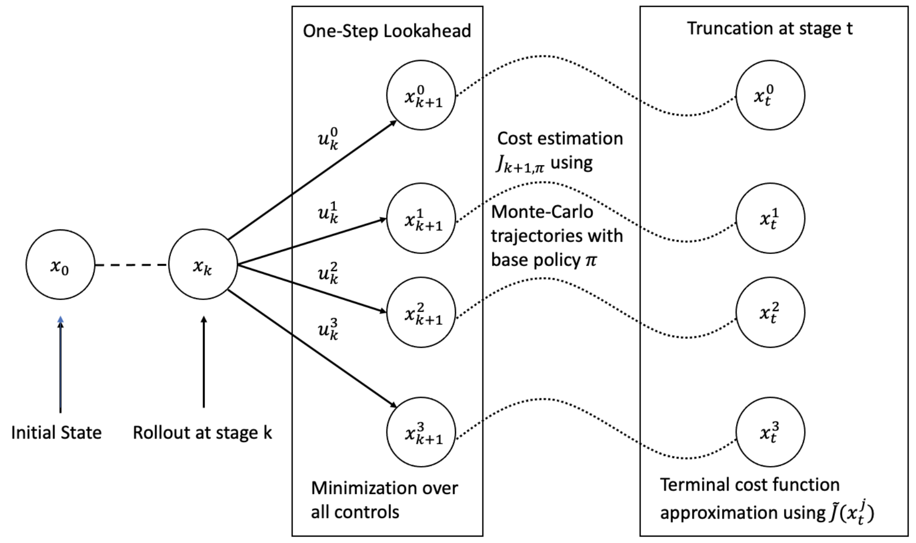


Figure 2.7: Rollout with One-Step Lookahead and Terminal Cost Function Approximation.

which we will then discuss in detail.

Generation of Policies from Values can be done by minimization of expected cost over all possible controls. Let us assume we have a cost function approximation $\tilde{J}$ for all states $s_i \in S$. Then we can derive a sub-optimal control policy at stage k, $\tilde{\mu}_k$ as:

$$\tilde{\mu}_k(x_k) = \arg \min_{u_k \in U(x_k)} \mathop{\mathbb{E}}_{w_k} \left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\} \tag{2.16}$$

Alternatively, if we have approximate q-factors $\tilde{Q}_k(x_k, u_k)$, then we can write its approximate policy as:

$$\tilde{\mu}_k(x_k) = \arg \min_{u_k \in U(x_k)} \tilde{Q}_k(x_k, u_k) \tag{2.17}$$

It can be seen that it is quite trivial to move from approximation in value or q-factor space to policy space.

Generation of values from policies by running multiple Monte-Carlo simulations using the policy π, especially in stochastic environments. At stage k, each simulation also known as a *trajectory* can be run to get the cost, and then an expectation can be taken over multiple such trajectories to find an approximate cost from that stage. We can say mathematically,

$$\begin{aligned} \tilde{J}_k(x_k) &= \mathbb{E}\left\{ J_{k,\pi}(x_k) \right\} \qquad \text{and,} \\ J_{k,\pi}(x_k) &= g_k(x_k, \mu_k(x_k), w_k) + J_{k+1,\pi}(x_k, \mu_k(x_k), w_k) \qquad \text{where } \mu_k \in \pi \end{aligned} \tag{2.18}$$

Notice here that in the case of an infinite horizon problem it is not possible to continue simulating the trajectories infinitely. Hence they are usually *truncated* after some stages (say stage t as shown in Fig. 2.7) and are replaced by an *approximate terminal cost*. This will be elaborated on in the next section.

Generation of new policies from values generated from policies - Rollout is a method that at stage k uses an l-step lookahead, in combination with the cost of the tail subproblem from $k + l^{th}$ stage to find the appropriate control for the stage. Mathematically, this can be written as:

$$\tilde{\mu}_k(x_k) = \arg \min_{u_k,\dots,u_{k+l-1}} \mathbb{E}\left\{ \sum_{i=k}^{k+l-1} g_i(x_i, u_i, w_i) + \tilde{J}_{k+l}(x_{k+l}) \right\} \tag{2.19}$$

This cost approximation from the $k + l^{th}$ stage can be computed using a base policy (or base heuristic), π, and estimating the cost using multiple Monte-Carlo trajectories. Hence, we can rewrite Eq. (2.19) as:

$$\tilde{\mu}_k(x_k) = \arg \min_{u_k,\dots,u_{k+l-1}} \mathbb{E}\left\{ \sum_{i=k}^{k+l-1} g_i(x_i, u_i, w_i) + J_{k+l,\pi}(x_{k+l}) \right\} \tag{2.20}$$

This resulting policy $\tilde{\pi} = \{\tilde{\mu_0}, \dots, \tilde{\mu}_{K-1}\}$ is called a *rollout policy.* If these heuristics follow the properties of *sequential consistency* and *sequential improvement*, we are guranteed to find that the rollout policy improves upon the base policy (defined by the base heuristic) i.e.,

$$J_{k,\tilde{\pi}}(x_k) <= J_{k,\pi}(x_k) \qquad \forall x_k \in S$$

The proof and the elaborate discussion can be found in [5] (Section 5.1.2), [14], and [8]. Since most greedy policies follow these properties, we can use rollout to improve upon the naive policies and get intelligent behaviors in the next iteration. This key idea forms the basis of the entire work. This property of rollout also makes it an ideal companion of one of the previously discussed algorithms i.e. *policy iteration*. We can use *rollout with policy iteration* to create a basic framework that can be applied to a plethora of reinforcement learning problems. We defer this analysis for now and will again pick it up in **Chapter 3**.

2.6 Monte Carlo (MC) Methods

We will talk about one of the approximation ideas that utilize a simulator of the environment to estimate the cost function J_π using a fixed arbitrary policy π by running multiple simulated trajectories from the initial state. We already introduced this idea in section 2.5.3. We showed that the cost can be estimated using Eq. (2.18) which is:

$$\tilde{J}_k(x_k) = \mathbb{E}\left\{ J_{k,\pi}(x_k) \right\} \qquad\qquad \text{and,}$$

$$J_{k,\pi}(x_k) = g_k(x_k, \mu_k(x_k), w_k) + J_{k+1,\pi}(x_k, \mu_k(x_k), w_k) \qquad \text{where } \mu_k \in \pi$$

Typically, our objective here is to estimate the cost of state $s_i \in S$ by running multiple simulated trajectories passing through the state s_i known as a visit to s_i. We average the cost returns through these visits and it can be shown that all these estimates converge to $\tilde{J}_\pi$. In a similar fashion, we can also use MC methods to estimate the *q-factors or action values* $\tilde{Q}_\pi(s_i, u)$. We can also use MC methods with *policy iteration* algorithm (alternating between policy evaluation and improvement) which is referred to as *Monte Carlo Control*.

Monte-Carlo methods are also amenable to parallel computing. This property makes them highly suitable for large problems as multiple trajectories can be simulated in parallel on a distributed cluster of machines. This method can also be combined with other methods like *DP*, *Temporal-Difference (TD) Learning* etc. These properties are the reason for using this method in our work on *rollout* methods.

However, one of the major challenges in RL and also with MC Control algorithms is the trade-off between *exploration* and *exploitation*. We want to find new optimal control and states but also want to exploit the already found optimal controls. Most of the MC Control algorithms use ϵ-greedy policies, which means that most of the time they select the best control (minimum cost control) but with probability ϵ they select any action randomly. More in-depth study and discussion can be found in [34]. In general MC methods work well for problems ranging from small state spaces to large ones like POMDP, but they do not provide any guarantees of performance. In the domain of POMDP solvers, there have been two major developments *POMCP* [30] and *DESPOT* [38] that use Monte-Carlo Tree Search (MCTS) to find optimal policy. These algorithms try to strike a favorable trade-off between *exploration* and *exploitation* in large POMDP search spaces, while also pruning the large state spaces.

2.6.1 POMCP

We will now discuss *Partially Observable Monte-Carlo Planning* (POMCP) [30] which is an algorithm that works on large POMDP by using Monte-Carlo tree search (MCTS) for belief update. The method uses sampling to break the *curse of dimen-*

sionality i.e. instead of exploring all state transitions focus on the best regions through efficient sampling. It also handles the *curse of history* by sampling the Monte-Carlo trajectories and using a black-box simulator of the environment.

The method uses MC simulation to evaluate the nodes of the search tree using the sequential best-fit order. The work claims that if the belief state is correctly chosen for a problem, this simple process can converge to optimal policy for any finite horizon POMDP. We now present the algorithm [6] as it is in [30].

Procedure ROLLOUT(s, h, depth)

```
Input: search tree $h$, state $s$, $depth$
if $\gamma^{depth} < \epsilon$ then
    return 0
$a \sim \pi_{rollout}(h, .)$
$(s', o, r) \sim G(s, a)$
return $r + \gamma.ROLLOUT(s', hao, depth + 1))$
```

Algorithm 3: Partially Observable Monte Carlo Planning

```
Input: search tree $h$
while $Timeout()$ do
    if $h == empty$ then
        $s \sim I$
    else
        $s \sim B(h)$
    end
    SIMULATE(s, h, 0)
end
return $\arg\max_b V(hb)$
```

[6] This ROLLOUT procedure used has a different meaning than what we use in the scope of this book. This is akin to the base policy used in this work, which is drastically different from our use of the word.

```
Procedure SIMULATE(s, h, depth)
    Input: search tree $h$, state $s$, $depth$
    if $\gamma^{depth} < \epsilon$ then
        return 0
    if $h \notin T$ then
        forall $a \in A$ do
            $T(ha) \leftarrow (N_{init}(ha), V_{init}(ha)), \phi)$
        return $ROLLOUT(s, h, depth)$
    $a \leftarrow \arg\max_b V(hb) + c\sqrt{\frac{\log N(h)}{N(hb)}}$
    $(s', o, r) \sim G(s, a)$
    $R \leftarrow r + \gamma.SIMULATE(s', hao, depth + 1))$
    $B(h) \leftarrow B(h) \cup \{s\}$
    $N(h) \leftarrow N(h) + 1$
    $N(ha) \leftarrow N(ha) + 1$
    $V(ha) \leftarrow V(ha) + \frac{R - V(ha)}{N(ha)}$
    return $R$
```

Note here that the algorithm uses different notations that are used in this work. The Monte-Carlo Tree is represented as h, with the node as T, belief state as B, observation as o, controls/actions as a, and reward (instead of cost) as R, the value (instead of cost) function is given by V. The works consider each node of the tree as a *multi-armed bandit* and utilize UCT algorithm [20] to improve upon the greedy selection in MCTS. In particular, the controls are chosen by the use of the UCB1 algorithm [3]. So, $N(h)$ and $N(h, a)$ which denote the visitation count of states, are used to control the exploration vs exploitation trade-off using the equation:

$$Q^{\oplus}(s, a) \leftarrow Q(s, a) + c\sqrt{\frac{\log N(s)}{N(s, a)}}$$

Belief updation is performed by choosing *Kalman filter particles* instead of a closed form belief update. The algorithm performs well in POMDP games like *POCMAN* (partially observable version of *PACMAN*), *battleship* and *rocksample.*

2.6.2 DESPOT

DESPOT [38] (an acronym that stands for *Determinized Sparse Partially Observable Tree*) was introduced as an improvement over its predecessor POMCP. In particular, the work aimed to improve the performance of MCTS by eliminating the UCT based exploration in POMCP and replacing it with a search on a sparse tree containing belief nodes reachable under K sampled observation scenarios. The method works by pruning the observation space to improve the running time of MCTS from $\mathcal{O}(|A|^D|Z|^D)$ to $\mathcal{O}(|A|^D K)$ where D is the depth of the tree, A is action space, Z is the observation space. They also provide some cost guarantees that depend on the value of chosen parameter K. The main algorithm known as AR-DESPOT uses two bounds $U(b)$ and $L(b)$ to facilitate the search at each node. The objective is to choose an observation that maximizes the *weighted excess uncertainty* $WEU(b')$ at each child node b'. If ϕ_b denotes the set of sampled observations at b, then

$$WEU(b') = \frac{|\phi_{b'}|}{|\phi_b|} excess(b')$$

where $excess(b') = U(b') - L(b') - \epsilon\gamma^{-\Delta(b')}$. Here Δ is the depth of tree starting from b under policy π, whereas ϵ and γ are constants. The algorithm presented in [38] is shown below.

Procedure RUNTRIAL(b,T)

```
Input: search tree $T$, belief node $b$
if $\Delta(b) > D$ then
    return $b$
if $b$ is a leaf node then
    expand b one lever deeper and insert all new nodes into T as children of b
$a^* \leftarrow \arg\max_{a \in A} U(b, a)$
$z^* \leftarrow \arg\max_{z \in Z_{b,a^*}} WEU(\tau(b, a^*, z))$
$b \leftarrow \tau(b, a^*, z^*)$
if $WEU(b) \geq 0$ then
    return $RUNTRIAL(b, T)$
else
    return b
```

Procedure BUILDDESPOT(b_0)

```
Input: initial belief node $b_0$
Sample a set $\phi_{b_0}$ of K random scenarios for $b_0$.
INSERT $b_0$ into $T$ as a root node.
while time permitting do
    $b \leftarrow RUNTRIAL(b_0, T)$
    forall $n \in$ nodes on path from $b$ to $b_0$ do
        backup $U(n)$ and $L(n)$
return $T$
```

Algorithm 4: Anytime Regularized - DESPOT

```
1 Set $b_0$ to inital belief.
2 while $TRUE$ do
3     $T \leftarrow BUILDDESPOT(b_0)$
4     Computer an optimal policy $\pi^*$ for T using DP eq. (4) in [38]
5     Execute the action $a$ of $\pi^*$
6     Receive observation $z$.
7     Update the belief $b_0 \leftarrow \tau(b_0, a, z)$
8 end
```

The belief updating step is similar to POMCP. The experiments were performed on *RockSample* problem with favorable results. We used both these methods for comparison on our POMDP sequential pipeline repair problem. We suggest the reader go through [38] for further study of this method.

2.7 Temporal-Difference (TD) Learning

Now we introduce a popular *model-free* RL paradigm that has been successful in robotics, learning to play Atari games, and was the first program to play the game of *Backgammon*. The Temporal-Difference method was used to develop the program *TD-Gammon* [36]. It uses the combination of both Monte-Carlo and DP methods. Like MC methods, TD can learn with its interaction with the environment, from experience (think of the real world as a simulator), whereas like DP, it updates estimates using previously known estimates of using a mechanism called *bootstrapping*.

To update its cost function estimate J_π of a learned/arbitrary policy π for state x_k, the process is not followed till termination, rather only the next few stages (like truncation in rollout). In its simplest form, known as $TD(0)$ or one-step TD the update rule is written as:

$$J_\pi(x_k) = J_\pi(x_k) + \eta\Big[g(x_k, \mu_k(x_k), w_k) + \alpha J_\pi(f_k(x_k, \mu_k(x_k), w_k)) - J_\pi(x_k)\Big], \quad \mu_k \in \pi \tag{2.21}$$

Here η is a step size, which makes it a constant-η MC and the rest symbols have their usual meaning. $g(x_k, \mu_k(x_k), w_k) + \alpha J(f(x_k, \mu_k(x_k), w_k))$ is known as *TD target* as it is known only after execution of the policy. The coefficient of η in eq.(2.21) is called *TD error* since it measures the error in estimates in relation to the previous known estimate.

One of the benefits of using TD methods is that it is an online method, for which modeling an environment is not a necessity, unlike both MC and DP methods. Also, it is known that both MC and TD methods converge asymptotically to the correct cost functions for a policy [34].

2.7.1 Q-Learning

Q-learning is an off-policy (does not require any arbitrary policy) reinforcement learning algorithm that uses the TD method to update the control-value or the Q-factors of state control pairs i.e. $Q(s_i, u)\ \forall u \in U(s_i), s_i \in S$. This algorithm has convergence guarantees to optimal q-factor Q^*, provided that the q-factors are updated for all pairs. The algorithm is presented below.

Algorithm 5: Q-learning Algorithm (Minimizing Cost)

```
Input: Step size $\eta$, small $\epsilon > 0$, $K$ =terminal stage
Initialize $Q(s_i, u)\ \forall u \in U(s_i), s_i \in S$ arbitrarily except $Q_K(,.) = 0$
while $True$ do
    $k = 0$ while $k < K$ do
        Choose $u_k$ from $U(x_k)$ using policy derived from $Q$ (eg. $\epsilon$-greedy)
        Take control $u_k$, observe cost $g_k$
        $x_{k+1} = f_k(x_k, u_k, w_k)$
        $Q(x_k, u_k) \leftarrow$
         $Q(x_k, u_k) + \eta[g_k(x_k, u_k, w_k) + \alpha \min_{u \in U(x_k)} Q(x_{k+1}, u) - Q(x_k, u_k)]$
        $k = k + 1$
    end
end
```

2.7.2 Deep Q-Network (DQN)

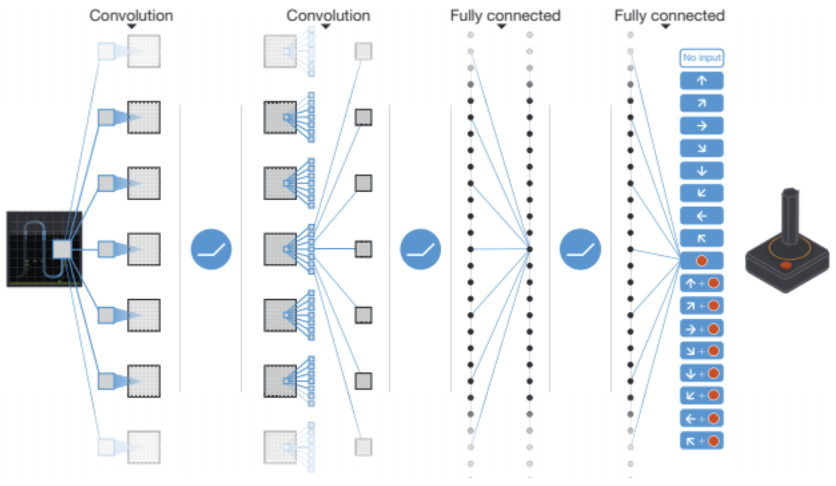


Figure 2.8: A DQN as Used in [24].

With the success of Deep neural networks, it has become imperative to use them as approximation architectures for various forms of learning methods described before. We had already introduced their applicability in both value and policy space approximation in sections 2.5.1 and 2.5.2 respectively. DQN was introduced in [24] [25] and have been particularly successful in playing Atari 2600 games using the raw pixels as inputs, intending to reach human-level performance.

There are two techniques that DQN uses for learning: *experience replay* and *target networks*. Experience is defined as a tuple (x_k, u_k, g_k, x_{k+1}). Similar to Q-learning at step k, the algorithm selects an ϵ-greedy control by looking at the q-factors and the entire experience is saved in a replay memory buffer that can store millions of transitions. Then a neural network as shown in Fig. 2.8 is trained on this data by using *stochastic gradient descent* to minimize the following loss:

$$L(\theta) = \left(g_{k+1} + \alpha \min_{u \in U(x_k)} \tilde{Q}_{\bar{\theta}}(x_{k+1}, u) - \tilde{Q}_{\theta}(x_k, u_k) \right)^2 \tag{2.22}$$

where θ denotes the parameters of a deep neural network and $\bar{\theta}$ are the parameters from the previous iteration. In case of the Atari games the state x_k was comprised of the sequence of image-control pairs denoted by $x_k = im_0, u_0, ..., u_{k-1}, im_k$. Also, preprocessing is performed by the application of function ϕ which is then fed to the neural network.

We now present the algorithm as shown in [24].

Algorithm 6: DQN training with Experience Replay (Minimizing Cost)

```
Input: Discount factor $\alpha$, small $\epsilon > 0$, $K$ terminal stage, $E$ episodes.
Initialize replay memory buffer $D$ to capacity $N$
Initialize $\tilde{Q}_\theta$ with random weights.
while $episode < E$ do
    $k = 0$
    Initialize sequence $x_0 = im_0$ and pre-processed sequence $\phi_0 = \phi(x_0)$
    while $k < K$ do
        Choose $u_k$ from $U(x_k)$ randomly with probability $\epsilon$
        otherwise select $u_k = \min_{u \in U(x_k)} \tilde{Q}(\phi_k(q_k), u; \theta)$
        Take control $u_k$, observe cost $g_k$ and image $im_{k+1}$
        $x_{k+1} = x_k, u_k, im_{k+1}$ and pre-process $\phi_{k+1} = \phi(s_{k+1})$
        Store transition/experience $(\phi_k, u_k, g_k, \phi_{k+1})$ in $D$
        Sample a random mini-batch of experiences $(\phi_j, u_j, g_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} g_j & \text{for terminal } \phi_{j+1} \\ g_j + \alpha \min_{u'} \tilde{Q}(\phi_{j+1}, u'; \theta) & \text{otherwise} \end{cases}$
        Run SGD on loss $(y_j - \tilde{Q}(\phi_j, u_j; \theta)^2$
        $k = k + 1$
    end
    $episode = episode + 1$
end
```

2.8 Policy Gradient Methods

Policy gradient methods are a family of approximate methods in policy space, developed for model-free reinforcement learning. Lately, these have been quite successful in various tasks including but not limited to Atari games, simulated robotics environments. Their salient feature is that they avoid dealing with cost functions of the state and hence can scale well to large state spaces especially since the policy space has been approximated. As discussed in section 2.5.2, we use parameters vector $\theta \in \mathbb{R}^{d'}$ to approximate the policy $\tilde{\pi}(u|s_i, \theta)$ such that at stage k we can write:

$$\tilde{\pi}(u|s_i, \theta) = \tilde{\mu}_k(x_k, \theta) = P(u_k = u|x_k = s_i, \theta_k = \theta) \tag{2.23}$$

where $u \in U(x_k)$, $s_i \in S$. If the control space is discrete then we can model the probability distribution in eq. (2.23) as a *softmax* function i.e.

$$\tilde{\pi}(u_i|s_i, \theta) = \frac{e^{h(s_i, u_i, \theta)}}{\sum_{j=0}^{|U|} e^{h(s_i, u_j, \theta)}} \tag{2.24}$$

which h is an action-preference function which could be linear/non-linear to the input features features. For the linear case we could write:

$$h(s_i, u_i, \theta) = \theta^T \phi(s_i, u_i)$$

Where $\phi \in \mathbb{R}^{d'}$ is an input feature vector. There can be three benefits of such formulation, the first being that we can reach a deterministic policy, unlike ϵ-greedy methods. The other is that softmax could potentially allow us to choose an action based on the probability distribution, which might be better for exploration in imperfect scenarios.

But there is a more important benefit of such formulation, the one that makes this methodology possible. The softmax function is differentiable and can be used to develop deep artificial neural network classifiers. For differentiable functions, we can define the gradient of the policy approximation as

$$\nabla \tilde{\pi}(u_i|s_i, \theta)$$

We now present the MC Policy Gradient control for policy π as derived in [34].

Algorithm 7: REINFORCE: MC Policy Gradient Algorithm

```
Input: Step size $\eta > 0$, a differentiable policy $\tilde{\pi}(u|s,\theta)$
Initialize $\theta$ policy parameter
while $True$ do
    Generate an episode $x_0, u_0, g_k, ..., x_{K-2}, u_{K-2}, g_{k-2}, x_{K-1}, g_{K-1}$
    $k = 0$
    while $k < K$ do
        $G = \sum_{j=k+1}^{K-1} \alpha^{j-k-1} g_j$
        $\theta = \theta - \eta \alpha^k G \nabla \ln \tilde{\pi}(x_k|u_k, \theta)$
        $k = k + 1$
    end
end
```

2.8.1 Actor Critic Framework

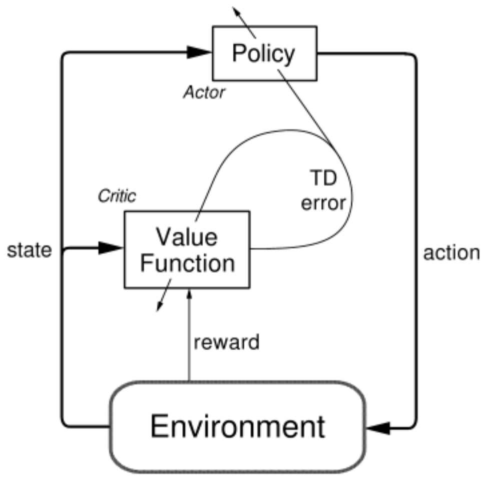


Figure 2.9: Actor-Critic Framework as Shown in [33].˘

The actor-Critic framework arose out of the ideas of both Policy Gradient and TD-methods. As the name suggests it consists of two components, the *actor* represents the parameterized policy as defined by eq. (2.24). The actor uses the parameters to approximate the policy, is used to select controls given the system state. The *critic* on the other hand is a cost function approximator, that uses *TD-error* as defined by equation

$$\delta_{TD} = g_k + \alpha J_{\tilde{\pi}}(x_{k+1}) - J_{\tilde{\pi}}(x_k) \tag{2.25}$$

to evaluate and improve the performance of the actor using the update equation below:

$$\theta = \theta - \eta\alpha^k \delta_{TD} \frac{\nabla \tilde{\pi}(x_k|u_k, \theta)}{\tilde{\pi}(x_k|u_k, \theta)} \tag{2.26}$$

Fig. 2.9 shows the architecture of actor-critic methods. There are two potential benefits of actor-critic methods over TD and Policy Gradient. They require minimal computation on control selection as unlike TD we don't need to update q-factors for all possible state-control pairs. Also, like PG methods, we can have a probabilistic action selection and a tendency to converge to deterministic policies. We now direct the reader to [34] for further analysis on this framework.

2.9 Multi-Agent Reinforcement Learning Algorithms

We now part with general RL and formally introduce the MARL paradigm. We already discussed that for MARL the MDP's are generalized to *Markov games.*

A *Markov game* is defined by a tuple $(M, S, \{U^\ell\}_{\ell \in M}, P, \{G^\ell\}_{\ell \in M}, \alpha)$. Let us assume there are m number of agents, then the $M = [1, ..., m]$ denotes the set of $m > 1$ agents, S denotes the state space observed by all the agents, control u consists

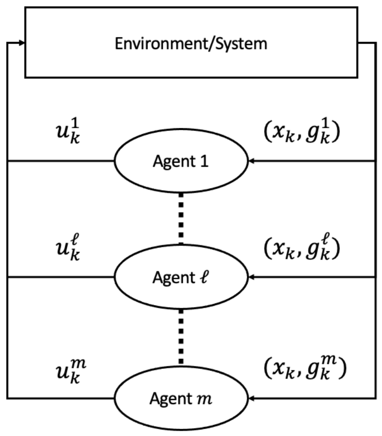


Figure 2.10: Illustration of Markov Games.

of m components, i.e. $u = (u^1, ..., u^m)$. Here each control component u^ℓ belongs to a finite set U^ℓ, which makes the control space a Cartesian product $U^1 \times U^2 \times ... \times U^m$. Also, $P : S \times U \times S \rightarrow [0, 1]$. The cost incurred at each stage by agent ℓ is given by $G^\ell : S \times U \times S \rightarrow \mathbb{R}$. Note the cost is dependent on the joint control space and not just individual control space of an agent. This coupling introduces additional challenges which were discussed in section 1.3.2. Fig. 2.10 shows the general Markov game. There is a detailed theory on these concepts which has been discussed in [39]. We will now discuss a few MARL algorithms relevant to this work.

2.9.1 Multi-Agent Deep Deterministic Policy Gradient (MADDPG)

MADDPG [23] is an actor-critic-based learning algorithm for MARL that aims to mitigate the challenges associated with MARL. The Q-learning-based methods suffer from *non-stationarity* whereas the policy-gradient methods fail to capture the variance that grows with the number of agents. To mitigate this actor-critic framework has been adapted to learn from the policies of other agents while successfully learning the policy that requires multi-agent coordination.

This work extends the DDPG framework, which utilizes a deep neural network for both actor and critic. The actors work locally and keep an estimate of the policies of other agents, the critic on the other hand could be centralized or decentralized depending upon the access to the global information. It is the centralized critic, that can essentially help agents develop coordination behavior during the training phase of the algorithm. Fig. 2.11 shows the training process. It can be seen that the actors are used only during the execution, whereas the critic is used to train the actors.

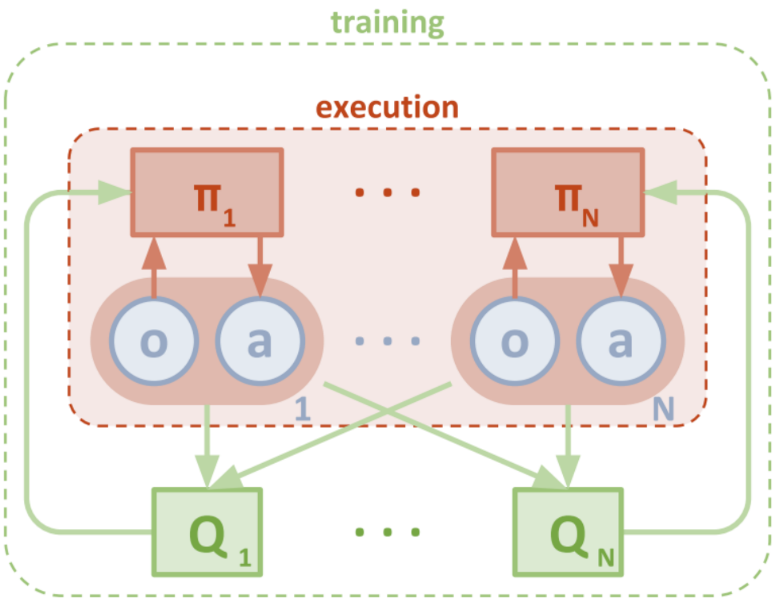


Figure 2.11: Illustration of MADDPG as Shown in [23].

2.9.2 QMIX - Monotonic Factorization of DQN for MARL

QMix [28] introduces a way to breakdown the Q-factors learned by a centralized DQN into individual agent DQN. This can be achieved in problems that follow the *monotonicity* property i.e.

$$\frac{\partial Q^{tot}}{Q^{\ell}} > 0 \quad \forall \ell \in M \tag{2.27}$$

Another way to put this is that the problem follows the following equation:

$$\arg\min_{u \in U(s)} Q^{tot}(s,u) = \left(\arg\min_{u^1 \in U^1(s)} Q^1(s,u^1), ..., \arg\min_{u^m \in U^m(s)} Q^m(s,u^m) \right)$$

For such problems, they introduced a special type of neural network called a *mixing network*, which takes in the q-factors from all m local DQN's and learns their monotonic mixing function by limiting the weights of the network to be non-negative. This is shown in Fig. 2.12. Note here τ is a pre-processed state s_i and o denotes the observation.

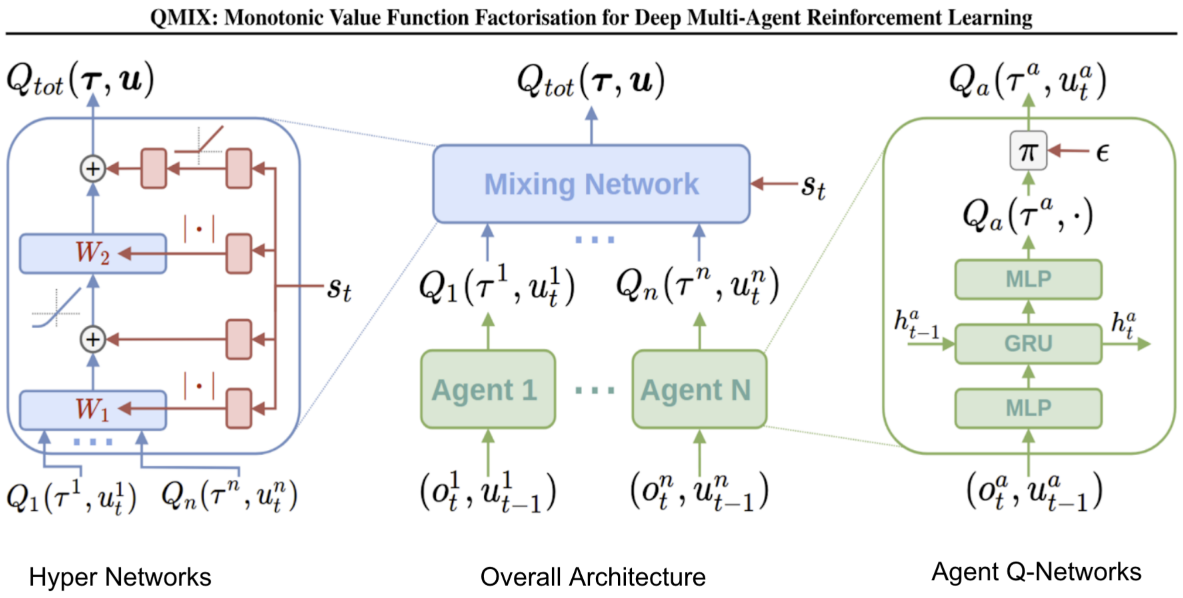


Figure 2.12: Illustration of QMIX as Shown in [28].

2.10 Summary

In this chapter, we formally introduce all basic algorithms that are important to understanding the algorithms, experiments, and results of this work. Note that the list of the algorithm is not exhaustive and it is possible that some major paradigms are not covered but that is because of their irrelevance to the work presented here. In the next section, we will talk about our methodology in detail and it will frequently require some of the concepts of partially observable, infinite horizon, stochastic DP problems, the MC methods, and Approximations discussed in this chapter.

Chapter 3

ROLLOUT METHODS

The rollout is an algorithm that utilizes an l-step look-ahead followed by a cost function approximation using a given base policy π to choose a control that minimizes the cost function, given the state (exact or partial) information and MDP parameters of the environment. This method has cost improvement properties and hence is then used in conjunction with *approximate policy iteration* scheme, using an approximation in policy and value space (use of neural networks). Our methods in this work were developed for infinite horizon discounted dynamic programming problems with finite state and control space and partial state observations. Although as we will see in the experiments section, they apply to a broader class of reinforcement learning problems. In this chapter, we present our methods [14],[15] as they were developed by building upon the ideas in the book [5].

We begin by revisiting our discussion on *Partially Observable Markov Decision Processes* (POMDP) in section 2.1.2. Recall that in absence of the perfect state information, we need to convert our problem into a perfect state information problem with the state represented as belief state $b = (b(s_0), ..., b(s_n))$, where $s_i \in S$ and $b(s_i)$ is the conditional distribution of state s_i given the prior history of observations $(z_0, u_0, ..., z_k, u_k)$, where $z_k \in Z$. It is a known fact that b is a sufficient statistic that can replace the history i.e. optimal controls can be taken with the knowledge of b.

Furthermore, we can generalize this belief state b to a *feature state* y which subsumes b.The reason for choosing y is that it can be enriched with additional information i.e. in addition to b, we can include a set of features that may be important to solving a problem. For instance the position of each agent in a multi-agent scenario. We also assume that this y can be generated in a sequentially using a *feature estimator* $F(y, u, z)$ i.e.

$$y_{k+1} = F(y_k, u_k, z_k) \tag{3.1}$$

The benefit of using y instead of b is that though the optimal policies arising out of b and y would be the same, y might facilitate the generation of better sub-optimal policies owing to the additional information. Additionally, the feature space y can help in achieving distributed computation using state-space partitioning as discussed in section 3.1.1.

Recall the equation 2.12, which states the optimal cost is given by the application of *Bellman Operator* T on the state s. Since, y is the sufficient statistic we can write:

$$(TJ)(y) = \min_{\mu \in \mathcal{M}} (T_\mu J)(y) \tag{3.2}$$

$$(T_\mu J)(y) = \hat{g}(y, \mu(y)) + \alpha \sum_{z \in Z} \hat{p}(z|b_y, \mu(y)) J(F(y, \mu(y), z)) \tag{3.3}$$

where b_y is the belief state corresponding to feature state y, T_μ is the Bellman Operator under policy μ, $\mathcal{M}$ is the set of all stationary policies and $\hat{g}$ denotes the expected cost which can be calculates as:

$$\hat{g}(y, u) = \sum_{i=1}^{N} b_y(s_i) \sum_{j=1}^{N} p_{ij}(u) g(s_i, u, s_j) \tag{3.4}$$

$\bar{p}(z|b_y, u)$ is the conditional probability that we observe z under belief b_y and control u, F as we already explained is the feature estimator.

As we discussed in section 2.5.1, we can introduce value space approximation in this formulation i.e. use an approximate estimate of cost $\tilde{J}$ instead of J. The sub-optimal policy under this approximation can then be written as

$$\tilde{\mu}(y) \in \arg\min_{u \in U} \left[\hat{g}(y,u) + \alpha \sum_{z \in Z} \hat{p}(b_y, u) \tilde{J}(F(y,u,z)) \right] \tag{3.5}$$

This equation is for a one-step lookahead scheme, but in an infinite horizon case, we could go for a more general l-step lookahead, where $\tilde{J}$ is the cost function of some base policy. We have already discussed this in our discussion of the rollout in the previous chapter (see Section 2.5.3).

3.1 Rollout and Approximate Policy Iteration (API)

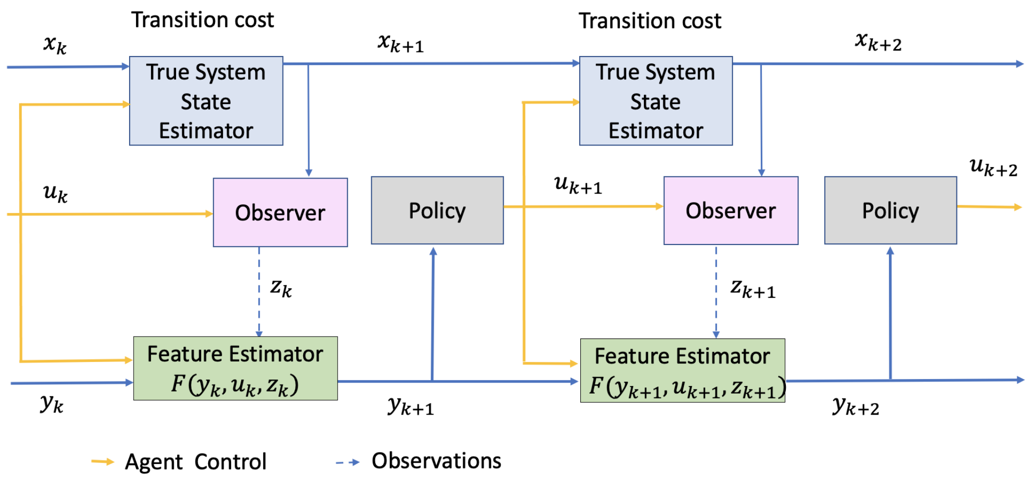


Figure 3.1: Illustration of POMDP Solver as Shown in [14].

We now focus our discussion on the pure form of rollout algorithm, wherein the cost function $\tilde{J}$ is the cost function of a known policy μ, called the *base policy* and its value $\tilde{J}(y) = J_\mu(y)$ which could be obtained in case of infinite horizon problems by running a Monte-Carlo simulator of the system model (see Fig. 3.1) from y (which subsumes b_y), under policy μ, using feature estimator F to some truncated horizon state $\bar{y}$ (which is reached after $\mathcal{K}$ steps) where we can use a *terminal cost function approximation* $\hat{J}(\bar{y})$ to yield the final cost of that simulated trajectory. Under a stochastic system, we would run multiple such trajectories to get an estimated cost of a rollout control. Once we have the estimated cost of all possible controls, we can choose the control that minimizes the cost using eq. (3.5). This was introduced in 2.5.3 and Fig. 2.7 illustrates the method and we now present the algorithm.

```
Procedure ROLLOUT($y, \mu, l = 1, \mathcal{K}, \hat{J}, \alpha$)
if $l == 0$ then
    run MC simulation using $T$ on $\mu$
    return $T_\mu^{\mathcal{K}} \hat{J}$
else
    Set $\tilde{u}$ arbitrarily
    $\tilde{J} = +\infty$
    for $u \in U$ do
        $\hat{g} = \sum_{i=1}^{N} b_y(s_i) \sum_{j=1}^{N} p_{ij}(u) g(s_i, u, s_j)$
        $t = 0$
        for $z \in Z$ do
            $p = \hat{p}(z|b_y, \mu(y))$
            $t = t + p * ROLLOUT(F(y, \mu(y), z), \mu, l-1, \mathcal{K}, \hat{J}, \alpha)$
        $J = \hat{g} + \alpha^l * t$
        if $J < \tilde{J}$ then
            $\tilde{J} = J$
            $\tilde{u} = u$
    return $(\tilde{u}, \tilde{J})$
```

Now we present without proof two *error bounds* associated with this methodology. Let us assume $\tilde{\mu}$ is the *rollout policy* generated by the application of the above-mentioned method. Then for a general l-step lookahead scheme, with truncated horizon after $\mathcal{K}$ stages, we have

$$||J_{\tilde{\mu}} - J^*|| \leq \frac{2\alpha^l}{1-\alpha}||T_\mu^{\mathcal{K}}\hat{J} - J^*|| \tag{3.6}$$

where $T_\mu^{\mathcal{K}}\hat{J}$ is the result of applying the Bellman operator T_μ, $\mathcal{K}$ times for policy μ and $||\cdot||$ is the sup norm on the space of bounded functions the feature state y. This implies the fact that the bound on the performance of rollout policy improves with the length l of lookahead.

Additionally, we have

$$J_{\tilde{\mu}}(y) \leq J_\mu(y) + \frac{2\alpha^{l-1}}{1-\alpha}||\hat{J} - J_\mu|| \quad \forall y \tag{3.7}$$

which implies that if $\hat{J}$ (terminal cost function approximation) and J_μ are close then the rollout policy $\tilde{\mu}$ improves upon the *base policy.* The proofs and the detailed discussion can be found in [5] (Section 5.1.2).

With these guarantees, the rollout can be used with *policy iteration* to obtain a method that consists of repeated *rollout* for policy improvement, and thus PI can be viewed as a *perpetual rollout process.* The idea is to save the rollout policy of the previous iteration, and use it as a base policy for the next iteration. For problems with large state spaces, we can use the approximation power of neural networks and supervised learning to obtain an approximation in policy space which has been discussed in 2.5.2. Recall that we used $\mu(\hat{.}, \theta)$ to denote the parametric approximation of the policy. Within this framework too, we can easily approximate the rollout policy by first generating sufficient $\{y^1, ..., y^q\}$ samples states and obtaining the sub-optimal controls by the rollout method. Once we have the dataset, we can train a neural network for both policy and cost function approximation using *stochastic gradient descent* SGD for minimizing the cross-entropy and sum of square errors respectively. The approximation in value space may be required for *terminal cost function approximation* in the truncated horizon version of the rollout.

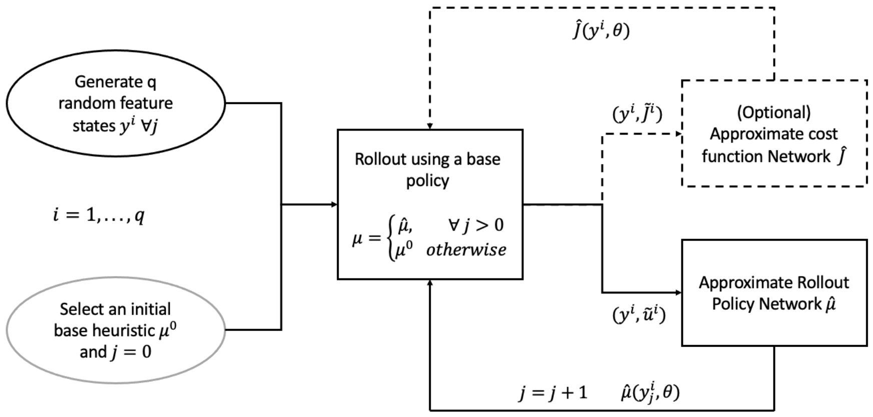

$\hat{J}(y^i, \theta)$
Generate q random feature states y^i $\forall j$
$i = 1, \ldots, q$
Select an initial base heuristic μ^0 and $j = 0$
Rollout using a base policy
$\mu = \begin{cases} \hat{\mu}, & \forall j > 0 \\ \mu^0 & otherwise \end{cases}$
$(y^i, \tilde{J}^i)$
(Optional) Approximate cost function Network $\hat{J}$
Approximate Rollout Policy Network $\hat{\mu}$
$(y^i, \tilde{u}^i)$
$j = j + 1$
$\hat{\mu}(y^i_j, \theta)$

Figure 3.2: Approximate PI Scheme Based on Rollout and Policy Space Approximation.

Since the approximation networks are only available after the first policy iteration, we need to choose the initial base policy and terminal cost approximations. For the base policy, we can choose any heuristic that can guarantee some form of *sequential consistency* or the very least *sequential improvement.* Most greedy heuristics follow at least one of these properties. The terminal cost function approximation can be either omitted i.e. *no truncation* (for finite-horizon problems) or we can simply use a constant cost (say zero) as an approximation. Fig. 3.2 shows the overall architecture of *approximate policy iteration.*

One noteworthy point is that like all RL algorithms, approximate PI schemes are also prone to issues relating to the exploration of the state space. If the sampling technique is biased or the sample size does not capture enough variance of the problem, we might end up getting bad approximations of policy and value. These issues are discussed in detail in [21],[17]. Our partitioned architecture (Section 3.1.1) tries to mitigate some of these issues. We now present our algorithm below:

Algorithm 8: Rollout API training

```
Input: Discount factor $\alpha$, lookahead $l = 0$, base heuristic $\mu$, truncation steps
       $\mathcal{K}$, $\hat{J}$ terminal cost function approximation, $\mathcal{E}$ policy iterations.
$\mu^0 = \mu$
$j = 0$
while $j < \mathcal{E}$ do
    Initialize dataset $D^p$ and $D^j$
    Generate $q$ random starting states $\{y^0, ..., y^q\}$
    foreach $y \in \{y^0, ..., y^q\}$ do
        $(\tilde{u}, \tilde{J}) = ROLLOUT(y, \mu^j, l = 1, \mathcal{K}, \hat{J}, \alpha)$
        add $(y, \tilde{u})$ to $D^p$ and $(y, \tilde{J})$ to $D^j$
    end
    Initialize parameters $\theta$ of a neural network classifier
    $\mu^{j+1} = TRAIN(\theta^p, D^p)$
    $\hat{J} = TRAIN(\theta^j, D^j)$
end
return $\mu^{\mathcal{E}-1}$
```

3.1.1 Partitioned Approximate Policy Iteration (PAPI)

Feature State Space Partitioning

In this section we introduce a method, that partitions the feature state space Y into disjoint sets $Y_1, ..., Y_N$, such that $Y = \bigcup_{\nu=1}^{N} Y_\nu$. We do so to train separate N (local) policy (and value) networks on these sub-sets. The N local policies approximated by these networks are then combined into a *global rollout policy*, defined over the entire feature space Y. We learn the local policies by using approximate policy iteration and a separate neural network for each local policy, using samples drawn from the corresponding subset of the feature space partition. The advantage of our partitioned architecture is that it can be trained in a piecemeal fashion and is well-suited for the use of distributed computation. We speculate that our methodology requires smaller training sets, which cover more evenly the feature space, thereby addressing in part the issue of adequate feature space exploration. This methodology is what we call *Partitioned Approximate Policy Iteration* (PAPI). There could be two different cases depending upon the problem or our choice to truncate the Monte-Carlo trajectories (shown in Fig. 3.3):

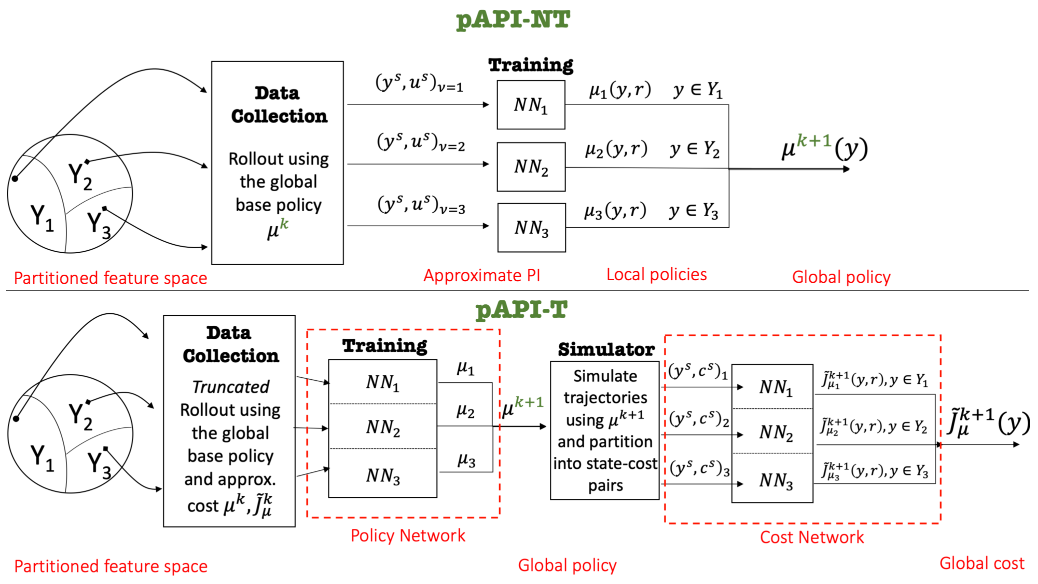


Figure 3.3: Partitioned Architecture for Rollout and Approximate PI with and Without Truncation [14].

1. *No Truncation* **(PAPI-NT)**, where each rollout policy is approximated using a policy network by the use of *rollout trajectories* with the large horizon (stages) (large enough for the discount factor α to make the cost of later stages

insignificant). Thus in this scheme, we do not need a value network for cost approximation. This might also be useful in the finite horizon problems, albeit maybe computationally more expensive.

2. *Truncated* **(PAPI-T)**, where the policy network is used to approximate a rollout policy generated by the use of *truncated rollout trajectories* that are approximated after $\mathcal{K}$ stages, by some cost approximation technique. The simplest way would be a *zero cost* or *constant cost* approximation, wherein we assume the cost to be some constant (say zero) after the truncation. A more sophisticated way could be the use of a neural network (value approximator) trained using the *global rollout policy.*

Asynchronous Computation

We now discuss the parallel computation aspect of our algorithm. As we already discussed, the partitioned architecture is amenable to the use of multiple processors that can work in parallel to compute the rollout policies for a feature space partition Y_ν. We present two methodologies that are based on partitioning and distributed computation. They involve one virtual processor for each set of the feature space partition (of course multiple virtual processors can coexist within the same physical processor).

How processors update and communicate their local policies and cost functions may lie between two extremes: "fixed sequential order", or "asynchronous". In *fixed sequential order*, the next iteration of *PAPI* at every processor waits for all processors to complete the previous iteration and then proceeds with the rollout for the next iteration. On the other hand in *asynchronous method*, each processor uses the latest communicated local policies and cost functions from other processors.

The obvious advantage of the asynchronous variant is that it does not incur a synchronization penalty, i.e. the cost of waiting for all other processors to complete and communicate their local computations before starting the computation of the next global policy. The processors simply use the information latest communicated from other processors, even if this information is out-of-date. The book [11] (Chapters 6 and 7) contains a detailed analysis of asynchronous distributed algorithms with examples. We also note that asynchronous distributed PI has been described in the papers [10] and [9], where proofs of convergence and error bounds of such methods involving partitions are given. Our partitioned algorithm also bears similarity with asynchronous distributed hard aggregation, which is described in [4] (Section 6.5.4).

3.2 Multi-Agent Rollout Methods

We now shift our focus to MARL and present the multi-agent variant of the rollout algorithm which was first conceptualized in [6]. As discussed in the Section 2.9, the major difference in MARL is the introduction of a joint control space U which is composed of m individual components i.e. $U = U^1 \times ... \times U^m$. As such one of the ways to solve MARL problems is to treat them as single-agent problems by treating the control u as a tuple $(u^1, ..., u^m)$. This is possible in cases where the number of controls and number of agents is small. It can be seen that the control space grows exponentially with the number of agents i.e. assuming $\mathcal{U} = \max\{|U^1|, ..., |U^m|\}$ then the number of controls per stage is given by $\mathcal{O}(\mathcal{U}^m)$. With this formulation, we can use all the algorithms discussed in chapter 2 as well as our rollout method discussed in previous sections. Although it is easy to see that all these methods might not scale well with the increase in the number of agents. This immediately calls for methods that can solve this scalability issue, without the loss of the cost improvement properties. We present these ideas in the following sections.

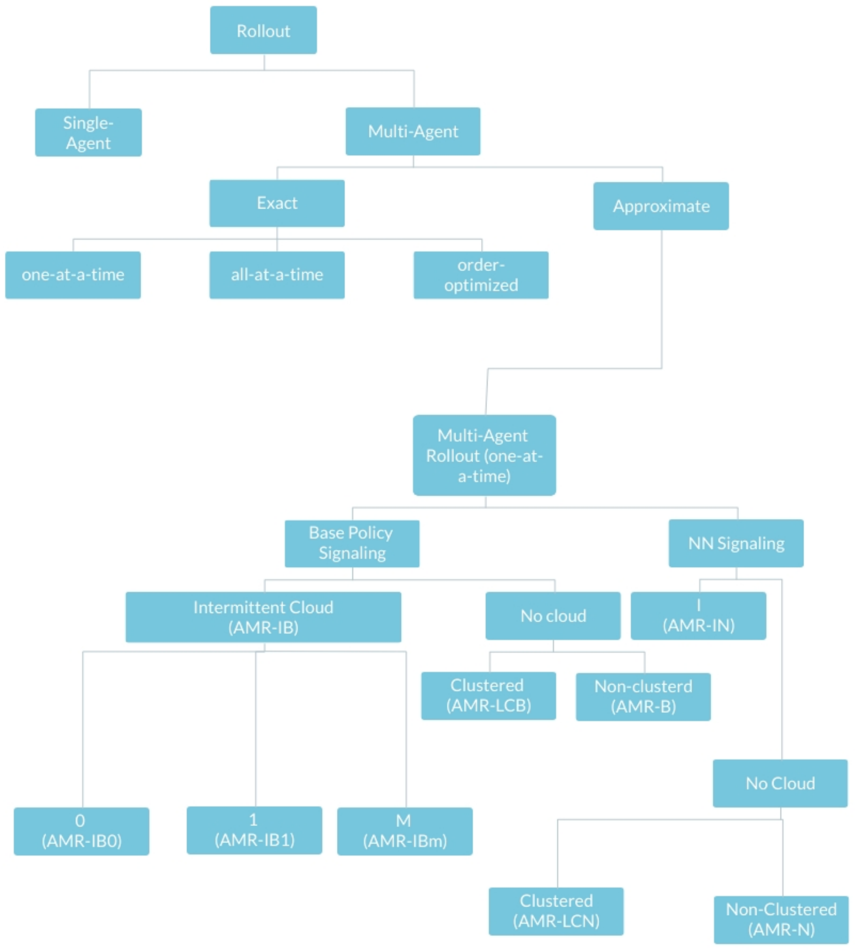


Figure 3.4: A Non-Exhaustive Taxonomy of Multi-agent Rollout.

3.2.1 Standard Rollout

We begin by first discussing the *standard rollout*, which is the application of the rollout algorithm in the joint control space. At each step k, we minimize over the set of all possible combinations of controls by first performing an l-step lookahead and then using the base policy, with or without truncation and terminal cost function approximation, to get an improved *rollout policy.* The difficulty with this method is the minimization complexity of $\mathcal{O}(\mathcal{U}^m)$ per stage (where $\mathcal{U} = \max\{|U^1|, ..., |U^m|\}$). To alleviate this, we present the next multi-agent rollout variant where we perform the aforementioned minimization one-agent-at-a-time. Fig. 3.5 shows the standard form of the rollout for two agents with the truncated horizon and a terminal cost function approximation.

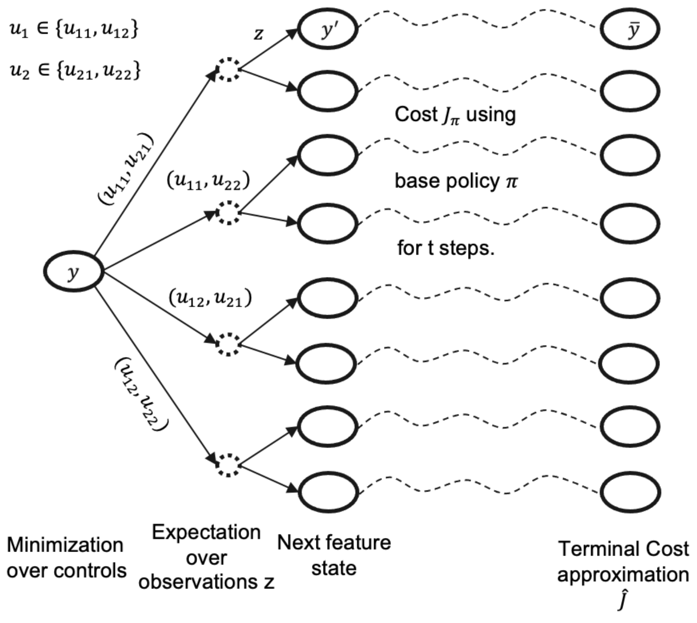


Figure 3.5: Standard Rollout with 2 Agents Adapted from [15].

3.2.2 One-At-A-Time Rollout

To reduce the minimization complexity, we need to simplify the formulation by converting it into another equivalent problem. Here the control $u = (u^1, ..., u^m)$ is broken down into m individual components i.e. given the feature state y, we generate m number of intermediate states (between y and next state y') by sequentially minimizing the control component for each agent and keeping all other controls constant. As discussed in [6], this can be thought of as a trade-off between control-space and state-space complexity. Thus, the transition between y and y' would contain intermediate states $\{(y, u^1), (y, u^1, u^2), ..., (y, u^1, ..., u^{m-1})\}$ assuming the agents choose their controls in a pre-defined order 1 through m. At the last transition from $(y, u^1, ..., u^{m-1})$ to y' we incur a cost $\hat{g}(y, u)$ of choosing control $u = (u^1, ..., u^m)$. The cost of other intermediary transitions is zero.

In this reformulation, we perform a total of m minimization, one over the control space U^ℓ of each agent ℓ, instead of one large minimization over the joint space U. This improves the per stage minimization complexity to $\mathcal{O}(\mathcal{U}m)$. It is also important to note that when minimizing over the Q-factors for agent ℓ, we set $u^1, ..., u^\ell$ to the rollout controls of earlier agents while set $u^{\ell+1}, ..., u^m$ to the base policy controls. So each agent uses the rollout controls of the predecessor while assuming the base policy controls of the agents next in order. Our experiments show that we can obtain substantial computations improvements while suffering from a marginal loss in performance. We direct the reader to [7], [8], that present the discussion and analysis of VI and PI in the context of one agent at time-based methods, where it is shown that this method maintains the cost improvement property and the error bounds are identical to the standard form of the rollout. Fig. 3.6 is the pictorial representation of this method.

3.2.3 Order-Optimized Rollout

One of the caveats of the *one-at-a-time* rollout was the assumption of a fixed and pre-defined order of execution of the overall minimization order m agents. We can improve the algorithm by relaxing this assumption by a trivial idea involving optimization of the agent order using a few extra minimizations. It involves assuming each agent as the potential candidate for the position in the order and then choosing

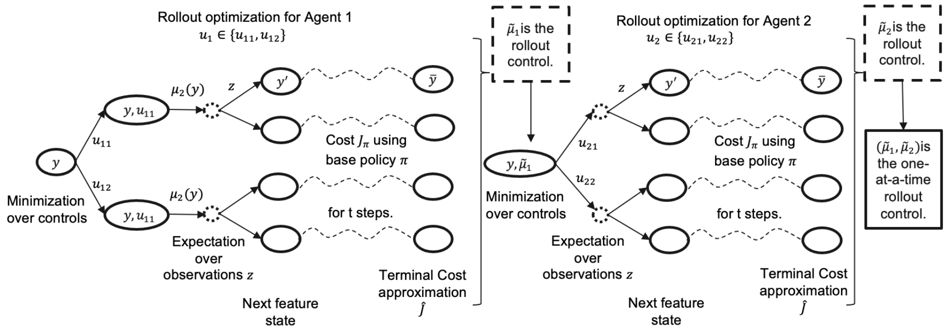

Rollout optimization for Agent 1
$u_1 \in \{u_{11}, u_{12}\}$
$\tilde{\mu}_1$ is the rollout control.
Rollout optimization for Agent 2
$u_2 \in \{u_{21}, u_{22}\}$
$\tilde{\mu}_2$ is the rollout control.
$(\tilde{\mu}_1, \tilde{\mu}_2)$ is the one-at-a-time rollout control.
Minimization over controls
Cost J_π using base policy π
for t steps.
Expectation over observations z
Terminal Cost approximation $\hat{J}$
Next feature state

Figure 3.6: One-At-A-Time Rollout with Agent 1 followed by Agent 2 adapted from [15].

the agent that has the minimal cost among all candidate agents. The position of the chosen agent is then fixed and it is removed from the candidate agents for the next position in the optimization order. For instance, for the first position, we would solve m minimization assuming each as a first agent in the order. Once an agent is selected based on the minimal cost, we can then remove it from the candidature for the second position that would involve $m - 1$ minimization, so and so forth. Once this process is carried on till the $m - 1^{th}$ position, we have the final optimal order of execution of one-at-a-time rollout. Thus, overall minimization increase from m to $m(m - 1)/2$ which though is a polynomial with degree 2, is still less than an exponential number of minimization per stage as in standard rollout. Our experiments validate that this process produces modest but consistent improvements over the vanilla *one-at-a-time* rollout. We now present both algorithms below.

Procedure MAROLLOUT$(y, \mu, l = 1, \mathcal{K}, \hat{J}, \alpha, u, \ell)$

```
1  if $l == 0$ then
2      run MC simulation using $T$ on $\mu$
3      return $T_\mu^{\mathcal{K}} \hat{J}$
4  else
5      Set $\tilde{u}$ arbitrarily
6      $\tilde{J} = +\infty$
7      for $c \in U^\ell$ do
8          $u[\ell] = c$
9          $\hat{g} = \sum_{i=1}^{N} b_y(s_i) \sum_{j=1}^{N} p_{ij}(u) g(s_i, u, s_j)$
10         $t = 0$
11         for $z \in Z$ do
12             $p = \hat{p}(z|b_y, \mu(y))$
13             $t = t + p * MAROLLOUT(F(y, \mu(y), z), \mu, l-1, \mathcal{K}, \hat{J}, \alpha, u, \ell)$
14         $J = \hat{g} + \alpha^l * t$
15         if $J < \tilde{J}$ then
16             $\tilde{J} = J$
17             $\tilde{u} = u$
18     return $(\tilde{u}, \tilde{J})$
```

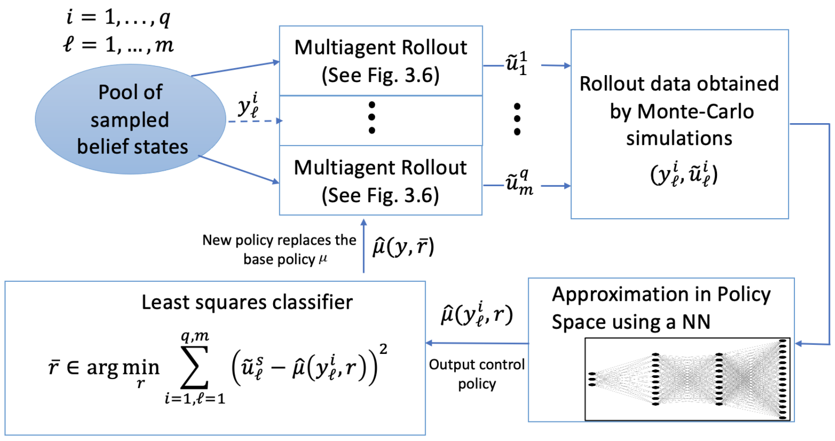
Algorithm 9: One-At-A-Time Rollout

```
Input: feature state $y$, base policy $\mu$, lookahead $l = 1$, truncated horizon
       after $\mathcal{K}$ steps, $\hat{J}$ ,$\alpha$, fixed order $M$ of agents
#initialize the controls to base policy controls
$u = (\mu(b_y)^1, ..., \mu(b_y)^m)$
for $\ell \in M$ do
    $(\tilde{u}, \tilde{J}) = MAROLLOUT(y, u, l = 1, \mathcal{K}, \hat{J}, \alpha, u, \ell)$
    $u = \tilde{u}$ #update the control of agent at $\ell^{th}$ position
end
return $(\tilde{u}, \tilde{J})$
```

```
Algorithm 10: Order-Optimized Rollout
Input: feature state $y$, base policy $\mu$, lookahead $l = 1$, truncated horizon
       after $\mathcal{K}$ steps, $\hat{J}$ ,$\alpha$, a set $M$ of agents
#initialize the controls to base policy controls
$u = (\mu(b_y)^1, ..., \mu(b_y)^m)$
Initialize a set $S = M$
while $|S| > 0$ do
    $J_{min} = +\infty$
    $bestAgent = 0$
    $u_{best} = u$
    for $\ell \in S$ do
        $(\tilde{u}, \tilde{J}) = MAROLLOUT(y, u, l = 1, \mathcal{K}, \hat{J}, \alpha, u, \ell)$
        if $\tilde{J} < J_{min}$ then
            $J_{min} = \tilde{J}$
            $bestAgent = \ell$
            $u_{best} = \tilde{u}$
        end
    end
    $u = u_{best}$
    $J = J_{min}$
    $S = S - \{bestAgent\}$#remove that agent from the set
end
return $(u, J)$
```

3.2.4 Multi-Agent Rollout with PAPI

Since the MA rollout methods do not change any basic assumption about the problem and its formulation and idea are largely the same, we can therefore use the same PAPI framework with all MA rollout variants. The process is almost the same, except the basic rollout blocks, are replaced by MA rollout blocks during policy iteration. Additionally, the feature state y may now include the location of each agent and a shared belief of the system. This is shown in Fig. 3.7.



$i = 1, \dots, q$
$\ell = 1, \dots, m$
Pool of sampled belief states
y_ℓ^i
Multiagent Rollout (See Fig. 3.6)
Multiagent Rollout (See Fig. 3.6)
$\tilde{u}_1^1$
$\tilde{u}_m^q$
Rollout data obtained by Monte-Carlo simulations $(y_\ell^i, \tilde{u}_\ell^i)$
New policy replaces the base policy μ
$\hat{\mu}(y, \bar{r})$
Least squares classifier
$\bar{r} \in \arg\min_r \sum_{i=1,\ell=1}^{q,m} \left(\tilde{u}_\ell^s - \hat{\mu}(y_\ell^i, r)\right)^2$
$\hat{\mu}(y_\ell^i, r)$
Output control policy
Approximation in Policy Space using a NN

Figure 3.7: MA Rollout with API [15].

3.3 Imperfect communication

Multi-Agent Rollout methods discussed above assume perfect communication of controls and beliefs among the agents. In real-world problems in robotics, it is seldom the case. We now present the methods for *approximate multi-agent rollout* (AMR) in an imperfect communication scenario i.e. where the agents do not share the local information (belief or controls) with other agents at all times, instead they estimate the likely control or belief of others. We call this estimation *signaling* [13] (as an agent though cannot directly communicate but can weakly signal using the system model and known policy network). We can also achieve higher parallelization and computational speedup as agents can act independently in the absence of communicated control. We present different communication architectures based upon these settings

We also consider two possible scenarios arising out of the imperfect communication case which is explained below:

1. **Imperfect communication of controls (AMR)**

   Such a scenario can arise in situations where the frequency of controls is very high and as such the agents can only communicate with each other after a few time-steps and probably within a specific distance from each other. We assume though that the belief is still shared through a cloud that is available to the agents at all times. This assumption may be valid as the belief communication only occurs after $m-1$ intermediate stages of one-at-a-time rollout and as such is less frequent than the control communication. Additionally, we also assume a case where the communication is restricted by the (r hop) distance between agents and the probabilistic communication in the presence of a communication relay cloud.

2. **Intermittent communication of belief states and controls (AMR-I)**

   Such a scenario assumes the agents cannot communicate the beliefs and controls to the other agents at all times. However, each agent perfectly knows its location and can observe its surroundings. It also has a local belief about the state of the system and the location of other agents. So, its local belief is an estimation of the global belief state, location, and controls of other agents. We also consider the presence of a centralized cloud, that acts as a buffer and relay of the messages to the agents. With a probability ρ the agents can receive the global belief state and thus can synchronize themselves. We use **AMR-I** as a suffix to such scenarios.

3.3.1 Signaling

We now discuss the signaling architectures [15] (Appendix) that we considered as a part of this work. But this list is not exhaustive and is also a work in progress. Fig. 3.4 presents an overall taxonomy of MA rollout including the approximate architectures being discussed.

Base Policy Signaling

The simplest form of signaling is the assumption that other agents are using their base policy controls. Mathematically, the control component of agent ℓ is given by:

$$\bar{u}^{\ell} = \arg\min_{u^{\ell} \in U^{\ell}} \hat{g}(b, u') + \alpha \sum_{zinZ} \hat{p}(z|b, u') J_{\mu}(F(b, u', z)) \tag{3.8}$$

where $u' = (u^{1:\ell-1}, u^{\ell}, u^{\ell+1:m})$ and u^k denotes the base policy controls of agent $k \neq \ell$. Recall that earlier the controls of previous agents were given by the rollout control. This assumption though crude can work surprisingly well in some initial conditions, on the other hand, may produce poor oscillatory behavior in some other conditions.

The Base policy signaling in the AMR case (where only controls are not communicated) is denoted as **AMR-B** whereas, in the AMR-I case, it is denoted as **AMR-IB**. In AMR-IB, one-at-a-rollout is performed when the belief when the cloud is available on the synchronized actual belief. Furthermore, in the case of AMR-IB, we explore three scenarios depending upon the number of rollout optimizations, 0, 1, or m performed during the independent execution phase (unavailability of cloud), which we denote by **AMR-IB0, AMR-IB1, AMR-IBm** respectively.

In AMR-IB0, an agent does not perform a one-at-a-time rollout at all during the time cloud is unavailable and just executes the base policy. In AMR-IB1, the agent performs only one rollout minimization assuming base policy controls for all agents, whereas, in AMR-IBm, the agent performs rollout for all m agents. Although it is important to state that our experimentation was limited to AMR-IB0 and AMR-IB1.

Neural Network Signaling

In this case, we use the policy networks for the estimation of the controls of other agents. In particular, we explored two schemes:

In **AMR-N**, m independent optimizations are performed, once over each agent's controls, wherein the predecessor's controls are given by the policy network (the latest policy iteration), but the controls of agents next in order are given by the base policy.

In **AMR-PI**, is similar to the previous case but the predecessor's controls are given by the best policy network (the best policy iteration), but the controls of agents next in order are given by the policy network of the previous iteration.

Local Communication Case (AMR-LC)

We also discuss an alternate scenario wherein the absence of a cloud, the agents can locally communicate with each other in a local distance (say r hops away in a graph). In such a scenario, we use the communicated rollout controls for agents within the r hop distance and assume base policy for all others. We use the acronym **AMR-LC** for this case as we assume shared belief in this experimentation. There could also be an unshared belief version of this problem but we did not explore this as a part of our experimentation.

Intermittent and Local Communication Case (AMR-ILC)

Finally, we also explore a merger of both cloud-based and local communication cases for a shared belief scenario and call it **AMR-ILC**. In this scenario, the belief and controls are shared in a r-hop distance with a probability of 1 but for agents farther away there is a probability ρ of communication at each stage. This architecture strikes a practical tradeoff as there is a possibility of rich information when a central cloud is available while at other times relying on the local information. Fig. 3.8 shows this setup.

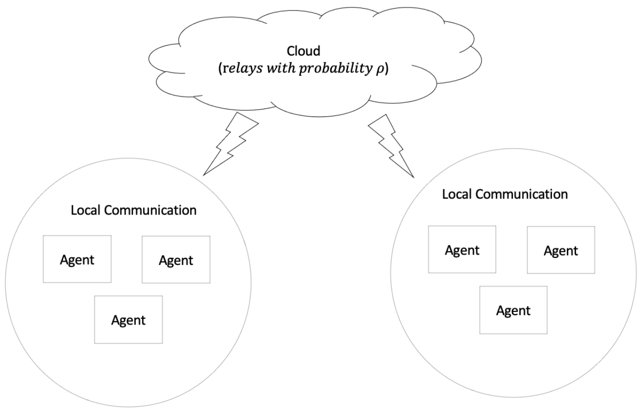


Figure 3.8: AMR-ILC with Local Communication and Unreliable Cloud.

3.4 Summary

This chapter provided a detailed overview of the methodologies developed as a part of this work. Our motivation for these problems lies in the practical applicability of RL, particularly MARL in real-world problems. There has been a huge success of RL in game playing. Starting from TD-Gammon [36] to Alpha Go [32] and Alpha Zero [31], the world has seen these algorithms involving Monte-Carlo simulation or *self-play* to reach new milestones in RL. Our work introduces methods that stem from mathematical guarantees of performance improvement, extended to include practical challenges like partial observations, stochasticity, etc., and are designed around the applicability to real-world problems. As we will see in the next chapter, we carefully choose three problems that are centered around real-world challenges and show that our methods find reasonable and sub-optimal solutions to these problems with little or no difficulty in even in practical challenges of stochasticity, intermittent communication, partial observation, etc.

Chapter 4

EXPERIMENTS AND RESULTS

4.1 Spider and Fly Problem

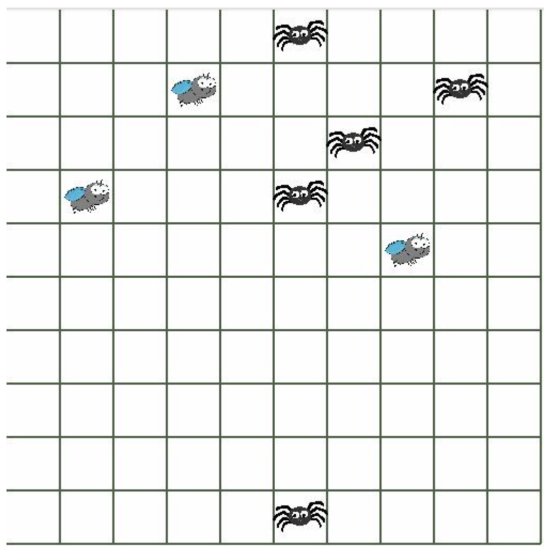


Figure 4.1: Spider and Fly Problem with 3 Flies and 5 Spiders.

Spider and Fly is a problem introduced in [7] which consists of a group of spiders that aim to catch a group of flies. The flies do not actively evade the spiders, rather move randomly in a network/grid. The problem is solved when all the flies are captured. Fig. 4.1 shows the spider and fly environment.

4.1.1 Motivation

The motivation of this problem arises out of search and rescue type problems where a group of search and rescue robots or drones can be deployed to search people or a group of people lost in a forest. The people might change position randomly while the search party is trying to rescue them. We might be able to observe their location at frequent times but they might be not aware and hence continue their random walk. Another motivation can be to control multiple fires in the forest/city. Fires can spread locally and thus have a dynamic movement (like flies). There too we might have an observation through drones/satellites but still might need a strategy to contain them as fast as possible.

4.1.2 Challenges

The challenges in this problem are twofold - one is the stochastic nature of the flies, which makes it difficult for each spider to plan their future course of action on the grid, second is the collaboration in the spider groups to create a strategy for capturing the group of flies. They should show behaviors like splitting to capture two distant flies or encircling to capture flies. We deliberately choose a perfect state observation in this scenario as it is a reasonable assumption in such problems.

4.1.3 Description

Our environment is a grid with both spiders and flies initialized to random locations. Each fly moves to one of the five possible locations at every time step (or stage) with a uniform probability distribution. For instance if the fly is on the location (i, j) at stage k, then it can be on five possible locations $\{(i+1, j), (i-1, j), (i, j+1), (i, j-1), (i, j)\}$ at stage $k+1$ with a uniform probability of $\frac{1}{5}$.

The state here is represented as the position of flies and spiders on the grid i.e $x =< pos(S), pos(f)) >$ where pos denoted the position, S and F are sets of spiders and flies respectively. The spiders can perfectly observe the state but during simulation, it has to deal with $5^{|F|+|S|}$ possibilities of observations.

There are five controls of spiders i.e. a unit step movement in any direction if permissible (*RIGHT, LEFT, UP, DOWN*) and choosing to stay at a position (*STAY*). At each time step, the spiders choose control from the five possibilities to catch the flies in minimum stages. The spiders capture the flies when they land on the same position in the grid and such flies are removed from the system.

We have chosen a *greedy* base policy wherein each spider moves towards the nearest un-captured fly and tries to catch it through direct pursuit in the direction of the fly. If the fly is captured, it pursues the next available fly. This policy is bound to solve the problem in an eventuality.

The cost at each stage is defined as the number of remaining flies. The average cost of the episode is given by the mean of the number of un-captured flies at each time step. The objective is to reduce the cost by capturing all flies as soon as possible.

4.1.4 Evaluation

We evaluate and compare the performance of our naive base policy with the rollout variants studied in section 3.2. We do so without the application of the policy iteration algorithm. We perform standard, one-at-a-time, and order-optimized rollout with 1-step lookahead and non-truncated rollout with the greedy base policy. The evaluation is performed over 1000 initial states, and we specifically observe the cost of each episode, number of steps required to capture all flies, and runtime per stage. We aim to validate our claims of performance improvement of standard as well as multi-agent rollout and also hope to see policies that reduce the cost by capturing the entire group of flies. In doing so we hope to observe natural behaviors like splitting and encircling.

4.1.5 Experiments

The objective of this experimentation is to show the cost improvement property of rollout and the efficacy of MA rollout to significantly improve the computation time while preserving the cost improvement property to an extent. We chose different grid sizes like 5×5, 10×10, and 20×20. We also vary the number of spiders and flies for each experiment (ranging from 2 spiders and flies to 8 spiders and 4 flies). We use a discount factor $\alpha = 0.99$ and use the same greedy base policy for all experiments and perform rollout without truncation. It is important to note that we do not parallelize the Monte-Carlo simulations because it is irrelevant in the comparison study (as all variants of the rollout have a similar method of Monte-Carlo simulation). The evaluation is performed on 1000 random initial states and the aggregated results are presented in the tables below. Each experiment is performed on an Intel Core i7-8565U Quad-Core CPU with 8 logical cores and 16 GB RAM. We now present our results and findings.

Table 4.1: Comparison of Base Policy, One-At-A-Time Rollout, Order-Optimized Rollout, and Standard Rollout for Spider and Fly Problem of Size 5X5, with 2 Spiders and 2 Flies on 1000 Random Initial States.

Method	Average Cost per Episode	Average Episode Length (stages)	Time (s)
Greedy	4.4362	5.384	4.8699×10^{-4}
One-At-A-Time Rollout	3.9514	4.878	**1.6976**
Order Optimized MA Rollout	3.9494	**4.864**	2.5364
Standard Rollout	**3.9404**	4.874	3.381

Table 4.2: Cost Comparison of Base Policy, One-At-A-Time Rollout, Order-Optimized Rollout, and Standard Rollout for Spider and Fly Problem of Size 5X5, with 3 Spiders and 3 Flies on 1000 Random Initial States.

Method	Average Cost per Episode	Average Episode Length	Time per Episode (s)
Greedy	5.7995	5.608	1.1556×10^{-3}
One-At-A-Time Rollout	4.7595	4.817	**4.064**
Order Optimized MA Rollout	4.6557	4.746	8.0112
Standard Rollout	**4.6168**	**4.737**	99.892

Table 4.3: Comparison of Base Policy, One-At-A-Time Rollout, Order-Optimized Rollout, and Standard Rollout for Spider and Fly Problem of Size 5X5, with 2 Spiders and 4 Flies on 1000 Random Initial States.

Method	Average Cost per Episode	Average Episode Length	Time (s)
Greedy	12.608	8.217	4.4677×10^{-3}
One-At-A-Time Rollout	10.3695	6.887	**18.5431**
Order Optimized MA Rollout	10.2796	6.865	27.6629
Standard Rollout	**10.1372**	**6.775**	580.5339

Table 4.4: Comparison of Base Policy, One-At-A-Time Rollout, and Order-Optimized Rollout for Spider and Fly Problem of Size 10X10, with 5 Spiders and 3 Flies on 1000 Random Initial States.

Method	Average Cost per Episode	Average Episode Length	Time (s)
Greedy	9.708	7.791	1.8934×10^{-3}
One-At-A-Time Rollout	8.8191	7.017	**32.198**
Order Optimized MA Rollout	**8.651**	**6.887**	94.9031

Table 4.5: Comparison of Base Policy, and One-At-A-Time Rollout for Spider and Fly Problem of Size 20X20, with 8 Spiders and 4 Flies on 1000 Random Initial States.

Method	Average Cost per Episode	Average Episode Length	Time (s)
Greedy	22.3423	12.611	1.03×10^{-2}
One-At-A-Time Rollout	**20.5451**	**11.102**	457.9651

1. **Validation of the cost improvement property**

   Through our experiments as seen in table 4.1, 4.2, 4.3, 4.4 and 4.5, we can see that in each experiment the rollout variants improve upon the greedy base policy. This empirical validation strengthens our claims of the efficacy of these methods. In each experiment, the Multi-Agent one-at-a-time rollout improves the cost. The cost improvement depends on the size and complexity of these experiments. Table 4.1 shows a 10.9% cost improvement of one-at-a-time rollout over greedy, whereas in table 4.3, a scenario with more flies than spiders, the cost improvement is 17.7%. Additionally, in the large experiment consisting of a grid size 20×20 as shown in table 4.5 we see a cost improvement by only 8% as the number of spiders is larger than the number of flies making the performance of greedy quite close but still worse than rollout.

2. **Comparison of Multi-Agent Rollout Variants**

   The experiments also validate our claims about the performance of rollout variants namely standard, one-at-a-time and order-optimized rollout. In each of the experiments, we see that the standard rollout performs the best followed by order-optimized and then one-at-a-time rollout. It is also important to note that the differences in performance are modest and hence even the worst-performing method (one-at-a-time) has significant cost improvements over base policy as discussed before. The average episode length also follows a similar trend with standard rollout taking minimum steps to catch all flies followed by the order-optimized and one-at-a-time rollout.

3. **Computational improvements of Multi-Agent Rollout vs Standard Rollout**

   The experiments also present the huge computational improvements achieved by using multi-agent rollout methods over the standard form. As seen in table 4.1, 4.2, 4.3, 4.4 and 4.5, the differences betweeen the standard and multi-agent methods become exponentially more prevalent. For instance for 2 spider case in 4.1 the time required for standard is approximately 2 times the time required for one-at-a-time rollout, but for the 3 spider problem in table 4.2 the difference is about 25 times.

4. **Parallel Rollout**

Table 4.6: System Specifications for the Experiment on Rollout Using Parallel Q-factor Computation.

CPU	Intel(R) Core(TM) i7-7700 CPU
Physical Cores	8
Hyper Threading/Core	34
Frequency	3.6 GHz
Cache	8192
RAM	32 GB

We also experiment with the parallelized version of the standard rollout on the problem to see the speedup achieved through the application of multi-processing. We compute each q-factor in parallel by implementing the algorithm using the BAIR's Ray [27] and use the system specifications as shown in table 4.6. Fig. 4.2 clearly shows that about 1.5x to 2x performance speedups can be achieved with the parallel q-factor computation per stage. We also show the normalized speedups achieved depending on the load average of the system. This is important as the system might be under different load settings throughout the experiment.

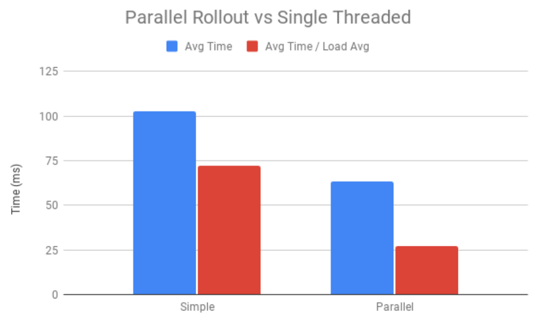


Figure 4.2: Results Showing Speedup Achieved Using Ray Vs a Single-Threaded Implementation of Rollout.

After showing these results we can now move to more complex problems. In particular, we discuss an infinite horizon, discounted POMDP with on a network-like environment, and also Flatland - a VRSP problem.

4.2 Multi-Robot Repair Problem

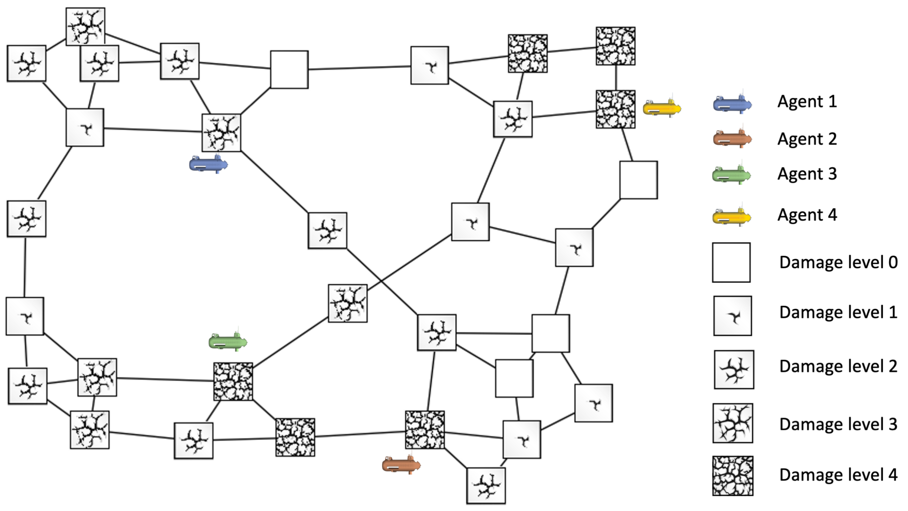


Figure 4.3: Multi-Robot Repair Problem.

Multi-Agent Robot repair problem was first introduced in [14] as an infinite horizon, discounted POMDP problem. It consisted of a linear pipeline with multiple damaged position which needed repair. The damages are divided into levels based on the degree of disrepair on the position. The damaged position can degrade over time if not repaired according to a known Markov chain. The agent tries to visit different positions, observe the damage level, and decides its further course of action based on its belief about the state of the environment. The objective here is to fix the pipeline as soon as possible. This problem was generalized to a graph structure and introduction of multiple robots in the subsequent work [15] (shown in Fig. 4.3).

4.2.1 Motivation

The motivation of such class of problems can be multi-robot repair tasks which include but are not limited to underground gas pipeline repair in ocean beds, forest fire management, cab services, etc. Damaged positions in the problem can be proxies for sites damaged in the underground pipeline, forest fire threats, cab pickup of a customer, etc. All these problems require effective coordination and intelligent behavior of agents to achieve the desired cost objective.

4.2.2 Challenges

The main challenges in this problem are partial observability, dynamic environments (changing damage levels), infinite horizon (the fixed position might fall into disrepair), and large control and state spaces. Hence, this represents a very rich class of problems that has the potential to be used as a benchmark problem in reinforcement learning.

4.2.3 Description

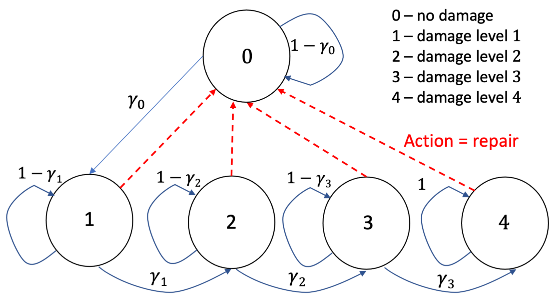


Figure 4.4: Markov Chain for Each Damaged Location From [15].

The problem consists of a network of a set of V nodes in an undirected graph, where each node has a damage level from a set $\nu = \{0, ..., \nu - 1\}$, where $\nu - 1$ is the highest damage level. These damaged nodes evolve according to a known Markov Decision Process (MDP) with $|\nu|$ states as shown in Fig 4.4. As we can see from the figure, a fixed state can fall into disrepair with a non-zero probability. This is a generalized extension of a linear pipeline problem introduced in [14]. The belief state can be represented as a tuple consisting of the belief of damage at each location. For instance, for a location v the belief is represented as $d^v = \{d_0^v, ..., d_{\nu-1}^v\}$. In case of m agents this is a POMDP with $|V|^m \cdot |\nu|^{|V|}$ states.

At each time step, once an agent at location v has observed its current location, it can either choose to stay in v and fix the location or move to one of its neighboring locations. Although the control space is fixed and small for a single agent, the joint control space is exponential to the number of agents is given by $|C|^m$ where C is the control set of a single agent.

The base policy was a *greedy policy* that similar to the spider and fly problem i.e. choose the nearest damaged location (irrespective of the damage level) and moves a step towards it. We used Dijkstra's shortest path algorithm to determine the shortest path from each location and consequentially the next hop.

The cost is incurred at each time step and is dependent on the state of the system. this is explained in the evaluation section below.

4.2.4 Evaluation

The algorithms are compared based on the average cost incurred by the algorithm in each episode. Each position is assigned a cost based on the damage level and the cost is defined as the weighted sum of all the damaged positions at each time step. Mathematically, at stage k it is given by $C_k = \sum_{\nu \in V} d^\nu \cdot c$, where c is a cost vector that assigns the weights to each level of damage. The policy needs to minimize the discounted sum of costs over an infinite horizon i.e. $\sum_k \alpha^k C_k$.

4.2.5 Experiments

The experimentation was performed on a graph topology as shown in Fig. 4.3 with 32 nodes. We used three sets of experiments with 4, 8, and 10 agents. This corresponds to the state space of 10^{28},10^{34}, 10^{37} and control space size of 625, $10^{5.6}$,

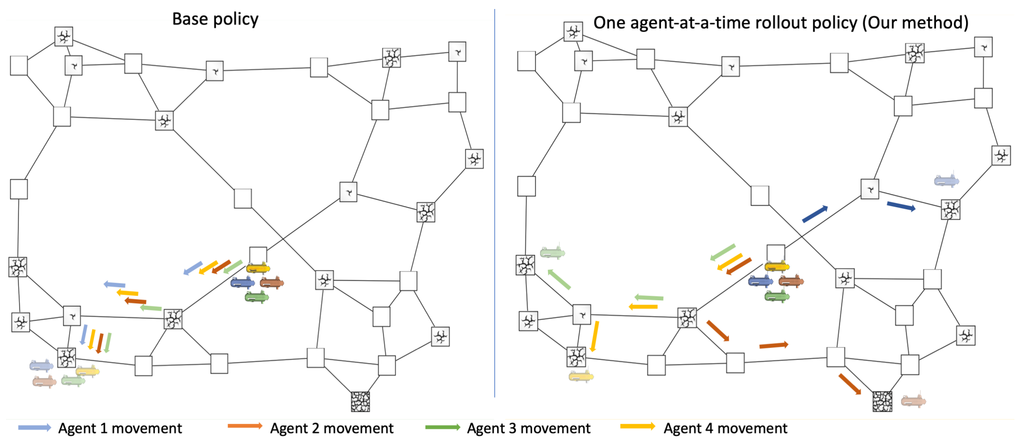


Figure 4.5: Comparison of Trajectories Generated by Greedy Base Policy and Rollout in the Multi-robot Repair Problem From [15].

10^7 respectively. The discount factor $\alpha = 0.99$ remained fixed throughout all these experiments. The MDP damage transition probabilities were fixed with $\gamma_0 = 0.01$ (0 for 4 agents), $\gamma_1 = 0.02, \gamma_2 = 0.03, \gamma_3 = 0.05, \gamma_4 = 0.1$. We perform two sets of experiments: one with the multi-agent rollout, and the other involving multi-agent rollout with API. Each experiment was performed on MPI enabled ASU Agave cluster with Intel Xeon E5-2680 v4 CPU (56, 196 cores respectively). Each evaluation is an aggregated result over 1000 random initial states. We now present the results.

1. **Cost Improvement of One-at-a-time rollout**

   Our experiments validate that significant cost improvement is achieved by the one-at-a-time rollout over the greedy base policy. Table 4.7 shows that our method consistently improves the cost achieved by the naive base policy. We also observe in Fig. 4.5 that where the greedy base policy results in all agents moving together towards the nearest fly, the rollout policy showing better behavior like splitting up to cover more damaged locations in parallel. Additionally, the rollout policy also showed prioritizing behaviors, wherein the agents decide to skip a nearby low damaged location in favor of a far highly damaged one.

Table 4.7: Cost Comparison of One-At-A-Time Rollout with Greedy Base Policy From [15].

Agents	Greedy Policy	One-at-A-Time Rollout
4	5347	992
8	4667	799

2. **Performance of MA Rollout with API**

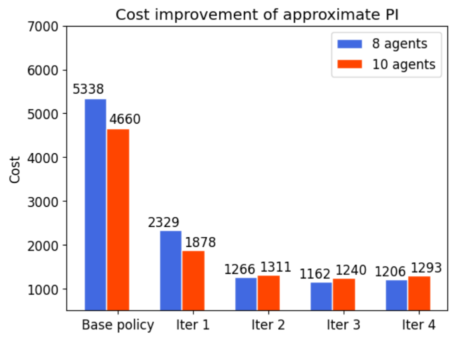


Figure 4.6: Cost Improvements in One-At-A-Time Rollout with API From [15].

For the approximate policy iteration version, we used a 2-layer fully-connected neural network with 256 and 64 ReLU units, followed by a batch normalization layer. The output is a softmax layer, which yields probabilities of the controls given a feature state. The size of the output of the layer is $|V|+1$ i.e. we predict the next hop of the agent for each $v \in V$ and one is added for the control of fixing the current location. We use RMSProp optimizer with a learning rate of 0.001 and use one-at-a-time rollout with 1-step lookahead and 10 simulated trajectories (to calculate cost expectation) for policy improvement at each policy iteration. This network is trained with 500,000 state-control pairs obtained through rollout in each iteration. These training samples contain a mix of randomly generated states, sampling from a previously generated set of states

- *a memory buffer*. The memory buffer contains the neighboring states of the previous iteration i.e. states reachable in a few stages from the previously (from the previous iteration) generated feature states. This is done to ensure a proper balance of exploration and exploitation which is one of the hard problems in any RL problem. Fig. 4.6 shows that the cost improvement property holds for several iterations even for a large state and control space.

3. **Performance comparison to other methods**

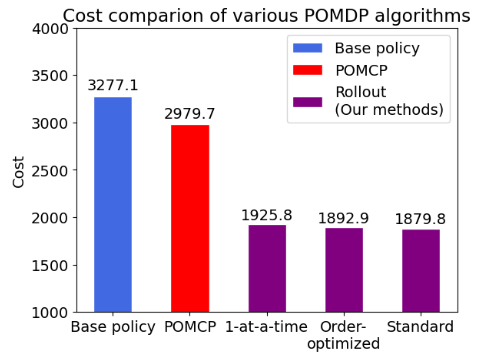


Figure 4.7: Cost Comparison of Greedy Policy, POMCP, and Other Multi-agent Variants of Rollout From [15].

We now compare the base policy with other variants of multi-agent rollout and several existing methods. In particular we explore Monte-Carlo methods like POMCP [30] and DESPOT [38], that are specifically developed to solve large POMDP. We also look at policy gradient methods for multi-agent scenarios like MADDPG [23] and a Q-learning-based method QMix [28]. Through our experiments we found that some of these methods do not scale well with the number of agents, hence we capped the number of agents to 4. Despite that, we found that DESPOT and QMix were unable to perform even better than the greedy base policy. POMCP was able to scale at a maximum of 4 agents and has been included in the results. Fig. 4.7 shows the comparison of these methods. We can see that as with the spider and fly problem, the rollout variants follow a similar trend with standard rollout being the best performing algorithm followed by order-optimized and one-at-a-time. But standard rollout could not scale up to more than 4 agents. It is important to note that all our methods performed significantly better than the base greedy policy.

POMCP method in our experiments performed slightly better than the base policy, at the same time suffering from scalability issues. We believe that the sparse lookahead of POMCP results in a poor estimation of Q-factors and hence it fails to perform like our methods. MADDPG on the other hand failed to perform even comparable to the base policy. We speculate that this might be because for such methods it is difficult to decide the hyper-parameters and the sample size to capture enough variance of the problem. It is again important to state that none of these methods provide any reasonable guarantees of cost improvement like our methods.

4. **Imperfect communication cases**

   We also experimented with the imperfect communication cases for our rollout-based methods i.e. *AMR* and *AMR-I*.

Table 4.8: Cost Comparison of Base, Standard Rollout (4 Agents Only), One-At-A-Time Rollout, and Different Approximate Multi-Agent Rollout Policies Involving Imperfect Control Communication (Assuming a Shared Belief) From [15].

agent	base	standard rollout	1-at-a-time	AMR-B	AMR-N	AMR-PI	AMR-LC r=2	AMR-ILC ρ=0.8, r=2	AMR-ILC ρ=0.5, r=2	AMR-ILC ρ=0.3, r=2
4	3277	1879	1925	3187	2635	-	2038	1946	1964	1976
8	5347	-	992	2513	1712	1590	1010	992	998	1005
10	4667	-	799	2487	1533	1428	813	804	807	809

Table 4.8 presents the results of different imperfect communication architectures involving unshared controls but shared a belief as discussed in section 3.3. We experiment with 4, 8, and 10 agents and run similar configurations for all these architectures. We observe that the effectiveness of the methods is dependent on the quality of signal or information used to estimate other agents' controls. Consequently, AMR-B performs the worst, even showcasing occasional oscillatory behaviors, because the base policy signal is not a correct estimation of other agents' control. AMR-N performs better than AMR-B as the policy network from the previous iteration is used as a signal, and AMR-PI is better than AMR-N due to better base and signal policies. AMR-LC works very well for our problem as it has a spatial structure that can be exploited by local communication i.e. Q-factors of local agents impact more than Q-factors of agents far apart. AMR-ILC works the best as it essentially the best of both worlds, i.e local and probabilistic global communication. But both these methods are highly dependent on the values of transmission probability ρ and transmission radius r. All these experiments validate that even in imperfect communication rollout-based methods can improve performance and generate intelligent behaviors.

Table 4.9 shows the performance of rollout methods involving both unshared belief and unshared controls in the presence of a cloud that shares information with the probability ρ. As seen in the table, the performance of these methods is dependent on the probability of the availability of complete information from the cloud. At $\rho \to 1$, the methods behave like one-at-a-time rollout while with $\rho \to 0$, behave like base policy. Another interesting observation is that AMR-IB0 (that assumes base policy) outperforms AMR-IB1, which performs a single rollout in the stages where the cloud is unavailable assuming base policy for other agents. This might be because AMR-IB0 does not perform any optimization in the absence of cloud and uses base policy controls till then.

Table 4.9: Cost Comparison of Base, One-At-A-Time Rollout, and Approximate Multi-Agent Rollout Policies with Different Intermittent Communication Architectures and Connection Probabilities (ρ) From [15].

agents	base	1-at-a-time	AMR-IB1 ρ = 0.8	AMR-IB0 ρ = 0.8	AMR-IB1 ρ = 0.4	AMR-IB0 ρ = 0.4
4	3277	1925	2303.96	2239.67	2793.45	2767.4
8	5347	992	1127.66	1140.21	1512.79	1713.17
10	4667	799	960.104	920.58	1265.49	1398.94

4.3 Flatland Challenge

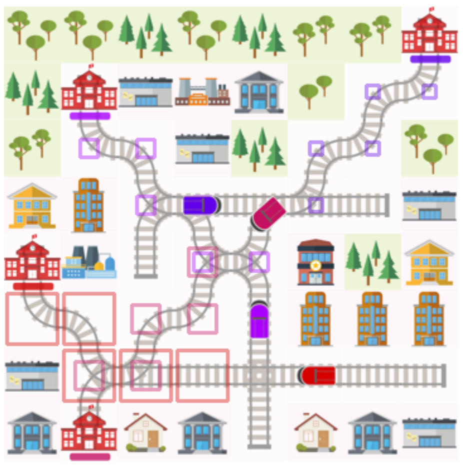


Figure 4.8: Flatland Environment with 4 Trains.

The Flatland Challenge [26] is an annual competition hosted on AICrowd and a part of conferences like NeurIPS 2020, AMLD 2021, to advance the progress in MARL for any re-scheduling problem (RSP). The problem is simplified on a 2D network of trains that need to reach their destination from a source with minimal delay. This addresses a major problem of transport and logistic companies i.e vehicle re-scheduling problem (VRSP) first studied in [22]. The central objective is to effectively manage dense traffic on a complex railway network in a stochastic environment (shown in Fig. 4.8).

4.3.1 Motivation

The vehicle rescheduling problem arises when the previously assigned trip is disrupted either due to a traffic accident/medical emergency, or a breakdown of a vehicle. The Swiss Federal Railways (SBB) manages 10,000 trains daily on a network of 13,000 switches and more than 32,000 signals. In the future, they need to increase the transportation capacity of approximately 30% on the same network and this is where efficient VRSP solutions are needed.

4.3.2 Challenges

This problem introduces two major challenges that make vehicle rescheduling essential. First is stochasticity, which means how often the trains will malfunction. This malfunctioning forces the agents to reconsider their plans (that can incur high costs). The second is speed profiles, which entails the different speeds for trains. This is an important factor, considering we would want to avoid scheduling a fast train behind a slow train. The other challenges may include scaling up to a large number of trains and large railway grid environments.

4.3.3 Description

The main goal is to make all the trains arrive at their target destination with minimal arrival time. The control space of each agent/train is one of the five controls:

- **DO_NOTHING**: If the agent is already moving, keep moving; if it is stopped, it stays stopped.

- **MOVE_LEFT**: The agent's head moves left if the left turn is possible. If the agent was stopped then it starts moving to its left, if taking left is possible.
- **MOVE_RIGHT**: The agent's head moves right if the right turn is possible. If the agent was stopped then it starts moving to its right, if taking right is possible.
- **MOVE_FORWARD**: The agent's head moves forward. If the agent was stopped then it starts moving in the forward direction.
- **STOP_MOVING**: The agent stops when this action is taken.

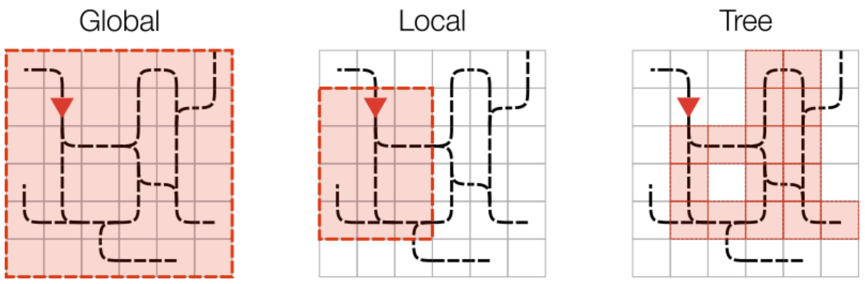


Figure 4.9: A Visual Summary of the Three Provided Observations [2].

The state for this problem is fully observable but can be re-modeled according to the solution. There are three types of observations provided - Global Observation that consists of all the grid information, Local Observation that only requires local information (a small grid around the agent's position), Tree Observation that represents the grids as the nodes reachable from the current position in the grid using the actions. It is also highly encouraged to create a custom observation depending upon the algorithm implemented. Fig. 4.9 shows all the three provided observations. We are currently using a Tree observation with an observation depth of 3.

The cost structure for an agent i consists of a local reward r_i (which is -1 id agent not at destination else 0) and global reward r_g (which is 1 if all agents reach their destination else 0). The cumulative reward is at stage k defined as $r_i(k) = \alpha r_l(k) + \beta r_g(k)$, where α and β control the collaborative behavior. The total and normalized cost of the episode for agent i is given by:

$$g_i = \sum_{k=0}^{K} r_i(k)$$
$$n_i = \frac{g_i}{K}$$

where K is the length of the horizon (number of steps to terminate the episode) for episode, which has a maximum upper limit depending upon the size of problem.

We choose multiple greedy policies for our experiments. We started with the shortest path greedy base policy for rollout as we feel it is a simple yet effective method. The only concern is that it might fall into deadlocks (i.e when two trains block each other). But we found that it is nevertheless a good base policy to start the experiments.

Since, it is a global challenge that has matured over the last few years, we are provided with some baseline algorithms like Dueling-Double Deep Q-Network (DDDQN) [37] and Proximal Policy Optimizations (PPO) [29]. As these methods use neural networks, we can use them as a base policy, with an objective to see some improvements by the use of MA Rollout.

4.3.4 Evaluation

The algorithm in the original challenge is evaluated with strict time limits (10s for each step) for planning and increasing the level of complexity and size of the environment.The objective is to gain the maximum reward across all environments.

For each task, the agents are evaluated on a fixed set of environment configurations ordered from low to high complexity. The solutions are evaluated using both the mean normalized reward, total normalized reward per agent and mean percentage completion (percentage of agents reaching destination) across all environments, which is calculated as follows :

$$\mathcal{G} = \sum_{j=0}^{N} (\frac{(\sum_{i=0}^{m_i} n_i^j)}{m_i} + 1)$$

$$\mathcal{N} = \frac{\sum_{j=0}^{N} (\frac{(\sum_{i=0}^{m_i} n_i^j)}{m_i} + 1)}{N}$$

where the scores ($\mathcal{G}$ and $\mathcal{N}$) are the accumulated total normalized reward and mean normalized rewards for N completed environment configurations with m_i agents for the respective configuration/episode. The solutions are ranked by $\mathcal{G}$ scores where a higher score is better.

Moreover, each submission is evaluated on the AICrowd servers making the comparison fair.

4.3.5 Experiments

1. **Experimentation with Shortest Path Base Policy**.

   We experiment with one-at-a-time rollout keeping $\alpha = 0.99$ with the greedy base policy. Table 4.10 shows that the cost improvement property still holds in this problem and hence we can move towards completing the next set of experiments.

2. **Experimentation with DDDQN Base Policy**.

   We experiment with approximate one-at-a-time rollout (**AMR-B**) keeping $\alpha = 0.99$ with the DDDQN baseline provided to us by the Flatland team. Table **??** shows that the cost improvement property still holds in this problem and hence we can move towards completing the next set of experiments.

Table 4.10: Cost Comparison of Shortest Path Base policy, and One-At-A-Time Rollout on Flatland.

Method	Mean Normalized Reward	Mean Percentage Completion
Shortest Path Base Policy	0.20	21.7
One-At-A-Time Rollout	0.47	50.0

Table 4.11: Cost Comparison of DDDQN Base Policy, and AMR-B on Flatland.

Method	Mean Normalized Reward	Mean Percentage Completion
DDDQN Base Policy	0.54	58.3
AMR-B	0.57	60.7

4.4 Summary

In this chapter, we formally introduced the three problems and the associated experimentation. Our results prove the cost improvement property and effectiveness of rollout-based methods in all these problems. In the next chapter, we conclude our work with a summary of the performance of our methods and also present a future direction for this research.

Chapter 5

CONCLUSION

5.1 Performance Summary

In this work we discussed various multiagent rollout methods and used them in an approximate PI scheme for challenging real-world problems involving stochasticity, partial obsersvability and an infinite horizon. We presented three different problems involving one or more of these challenges - Spider and Fly, Multi-Robot repair problem and Flatland challenge. In each of these problems, we verified the cost improvement property of multi-agent rollout variants, similar to standard rollout, with dramatically less computation and communication requirements. Similarly, we showed that multiagent approximate PI improves the policy at each iteration in order to find the sub-optimal policy, that is better then the base policy. The agents executing the resulting policy achieve a high degree of coordination with each other and showcase intelligent behaviors like splitting up the work, surrounding a target to catch it, and handling frequent changes in environment etc. We also reported numerical performance results on some imperfect communication extensions of our multiagent rollout methods. Our experimentation posits that these methods work well for robotics problems especially when a large team of multiple robots need to collaborate on a complex task with one or more of aforementioned challenges.

5.2 Future

We now present some further directions for our work. These directions arise out of the possible caveats of our methodology which is the online component of rollout that can be computationally expensive as discussed in the Flatland experimentation. Using parallel monte-carlo simulation would enable us to apply this methods to settings where there is a need for real-time response from the agent. We believe that any algorithm that does not re-plan (for instance a policy network) cannot deal with rapidly changing environments unlike rollout. We now discuss these caveats and possible directions one by one.

1. Multi-Agent Rollout with PAPI

   One of the extension to our work could be the implementation of MA rollout variants on the partitioned architecture. We introduced the partitioned approximate policy iteration (PAPI) in [14], that features the partitioning of the feature space depending on the problem. This partition enables us to exploit distributed computation by the use of multiple policy networks (for instance one for each partition) that can be trained in parallel. We can use this architecture for our multi-agent rollout variants as well and achieve desired parallelization.

2. Extensive experimentation on imperfect communication cases

   We can also perform extensive experimentation with the imperfect communication and possibly derive some cost improvement guarantees in such scenarios. We have already seen that such guarantees do not just serve the theoretical purpose but can be use to advance the methodology itself. Future of robotics lies in the success of such methods that are robust to imperfect communication scenarios.

3. Experimentation on real-world problems with real datasets.

   Another possible direction is the use of real-world datasets to solve an optimization problem. One such example could be improving the Cab service using reinforcement learning. Many open source datasets are available which can be used to solve such problems. One immediate benefit of solving this problem is that we can then deploy our algorithm in the real-wold settings and understand more about their applicability. The focus of this research has been to make the use of reinforcement learning ubiquitous in the optimization problems in the world, which is currently dominated by supervised and semi-supervised learning that do not work well which changing environments.

4. Using Aggregation for approximation in problem space

   Aggregation methods [5] are approximations in the problem space, which can be used to improve the performance of many rollout methods by reducing the complexity in the state space. This can be useful for some problems that are amenable to such approximation. These ideas have not been tried on real-world problems discussed in this work and hence it is a potential direction to explore.

5. Development and release of parallel rollout framework.

   This is possibly the most important direction as the rollout methods suffer from the curse of simulation. It is computationally expensive to run multiple rollout trajectories to calculate the expectations. With the current state of parallel computation softwares, I believe it should not be a challenging task. In fact a groundwork for this has been achieve in our implementation of rollout algorithm for the Flatland challenge. We also hope to release the parallel version of rollout to the public, so that it can be applied to multiple applications.

We conclude by noting that most of this work has been done on these three problems. Like any research, there cannot be a general claim of effectiveness of these methods to all problems. But our experiments do verify that for some realistic environment settings and even with a simple base policy, we can get guarantee performance improvements which is typically not found in the other existing work in this field.

REFERENCES

[1] Achiam, J., “Spinning Up in Deep Reinforcement Learning”, URL `https://spinningup.openai.com/`.

[2] AICrowd, “Provided Observations in Flatland”, URL `https://flatland.aicrowd.com/getting-started/env/observations.html`.

[3] Auer, P. and Cesa-Bianchi, N. and Fischer, P., “Finite-Time Analysis of the Multiarmed Bandit Problem”, Machine learning **47**, 235–256.

[4] Bertsekas, D. P., *Dynamic Programming and Optimal Control* (Athena Scientific, 1995).

[5] Bertsekas, D. P., *Reinforcement Learning and Optimal Control* (Athena Scientific, 2019).

[6] Bertsekas, D. P., “Multiagent Rollout Algorithms and Reinforcement Learning”, arXiv preprint arXiv:1910.00120, 2020 .

[7] Bertsekas, D. P., “Multiagent Value Iteration Algorithms in Dynamic Programming and Reinforcement Learning”, Results in Control and Optimization **1**, 100003.

[8] Bertsekas, D. P., *Rollout, Policy Iteration, and Distributed Reinforcement Learning* (Athena Scientific, 2020).

[9] Bertsekas, D. P. and H. Yu, “Q-Learning and Enhanced Policy Iteration in Discounted Dynamic Programming (Revised)”, Tech. rep.

[10] Bertsekas, D. P. and H. Yu, “Q-Learning and Enhanced Policy Iteration in Discounted Dynamic Programming”, Math. Oper. Res. **37**, 1, 66–94.

[11] Bertsekas, D. P. and Tsitsiklis, J. N., *Parallel and Distributed Computation: Numerical Methods* (Prentice-Hall, 1989).

[12] Bertsekas, D. P. and Tsitsiklis, J. N., “Neuro-Dynamic Programming: An Overview”, in “Proceedings of 1995 34th IEEE conference on Decision and Control”, vol. 1, pp. 560–564 (IEEE, 1995).

[13] Bertsekas, D.P., “Multiagent Reinforcement Learning: Rollout and Policy Iteration”, IEEE/CAA Journal of Automatica Sinica **8**, 2, 249–272.

[14] Bhattacharya, S., Badyal, S., Wheeler, T., Gil, S. and Bertsekas, D. P., “Reinforcement learning for POMDP: Partitioned Rollout and Policy Iteration with

Application to Autonomous Sequential Repair Problems", IEEE Robotics and Automation Letters **5**, 3, 3967–3974.

[15] Bhattacharya, S., Kailas, S., Badyal, S. and Gil, S. and Bertsekas, D. P., "Multiagent Rollout and Policy Iteration for POMDP with Application to Multi-Robot Repair Problems", arXiv preprint arXiv:2011.04222, 2020 .

[16] Cybenko, G., "Approximation by Superpositions of a Sigmoidal Function", Mathematics of Control, Signals and Systems **2**, 4, 303–314.

[17] Dimitrakakis, C. and Lagoudakis, M. G., "Rollout Sampling Approximate Policy Iteration", Machine Learning **72**, 3, 157–171.

[18] Dulac-Arnold, G. and Mankowitz, D. and Hester, T., "Challenges of Real-World Reinforcement Learning", arXiv preprint arXiv:1904.12901, 2019 .

[19] Hornik, K., "Approximation Capabilities of Multilayer Feedforward Networks", Neural Networks **4**, 2, 251–257.

[20] Kocsis, L. and Szepesvári, C., "Bandit Based Monte-Carlo Planning", in "Machine Learning: ECML 2006", pp. 282–293 (Springer Berlin Heidelberg, 2006).

[21] Lagoudakis, M. G. and Parr, R., "Reinforcement Learning as Classification: Leveraging Modern Classifiers", in "Proceedings of the 20th International Conference on Machine Learning (ICML-03)", pp. 424–431 (2003).

[22] Li, J-Q., Mirchandani, P. B. and Borenstein, D., "The Vehicle Rescheduling Problem: Model and Algorithms", Networks: An International Journal **50**, 3, 211–229.

[23] Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P. and Mordatch, I., "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments", arXiv preprint arXiv:1706.02275 [cs.LG], 2020 .

[24] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., "Playing Atari with Deep Reinforcement Learning", arXiv preprint arXiv:1312.5602, 2013 .

[25] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G. *et al.*, "Human-Level Control through Deep Reinforcement Learning", nature **518**, 7540, 529–533.

[26] Mohanty, S., Nygren, E., Laurent, F., Schneider, M., Scheller, C., Bhattacharya, N., Watson, J., Egli, A., Eichenberger, C., Baumberger, C., Vienken, G., Sturm, I., Sartoretti, G. and Spigler, G., "Flatland-RL : Multi-Agent Reinforcement

Learning on Trains", arXiv preprint arXiv:1312.5602 [cs.AI], 2020 .

[27] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W. and Jordan, M. I. *et al.*, "Ray: A Distributed Framework for Emerging AI Applications", in "13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)", pp. 561–577 (2018).

[28] Rashid, T., Samvelyan, M., Schroeder de Witt, C., Farquhar, G., Foerster, J. and Whiteson, S., "QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning", arXiv preprint arXiv:1803.11485 [cs.LG], 2018 .

[29] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., "Proximal policy optimization algorithms", arXiv preprint arXiv:1707.06347, 2017 .

[30] Silver, D. and Veness, J., "Monte-Carlo Planning in large POMDPs", (Neural Information Processing Systems, 2010).

[31] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. *et al.*, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", arXiv preprint arXiv:1712.01815, 2017 .

[32] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. *et al.*, "Mastering the Game of Go without Human Knowledge", Nature **550**, 7676, 354–359.

[33] Sutton, R., "Actor-Critic Methods", URL `http://incompleteideas.net/book/first/ebook/node66.html`.

[34] Sutton, R.S. and Barto, A.G., *Reinforcement Learning: An Introduction*, Adaptive Computation and Machine Learning series (MIT Press, 2018).

[35] Szepesvári, C., "Algorithms for Reinforcement Learning", Synthesis Lectures on Artificial Intelligence and Machine Learning **4**, 1, 1–103.

[36] Tesauro, G., "Temporal Difference Learning and TD-Gammon", J. Int. Comput. Games Assoc. **18**, 88.

[37] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. and Freitas, N, "Dueling Network Architectures for Deep Reinforcement Learning", in "International Conference on Machine Learning", pp. 1995–2003 (PMLR, 2016).

[38] Ye, N., Somani, A., Hsu, D. and Lee, W. S., "Despot: Online POMDP Planning with Regularization", Journal of Artificial Intelligence Research **58**, 231–266.

[39] Zhang, K., Yang, Z. and Başar, T., “Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms”, arXiv preprint arXiv:1911.10635, 2019 .