Autonomous Decision-Making Schemes for Real-World Applications in Supply Chains and Online Systems

Mohammadreza Nazari
Lehigh University

Autonomous Decision-Making Schemes for Real-World Applications in Supply Chains and Online Systems

by

Mohammadreza Nazari

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Industrial and Systems Engineering

Lehigh University

August 2019

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Date

Dissertation Advisor

Committee Members:

Lawrence V. Snyder, Committee Chair

Martin Takáč, Committee Co-Chair

Alexander L. Stolyar

Katya Scheinberg

Mustafa Kabul

Acknowledgements

I owe a debt of gratitude to a great number of people for helping me reach the culmination of this work. This dissertation is the result of the teachings of many great professors, the discussions and suggestions from incredibly intelligent and patient mentors and the encouragement and unconditional love from my friends and family.

First and foremost, I would like to sincerely acknowledge my advisors. I would like to express my gratitude to my advisor Professor Snyder for his endless support. With his patience, motivation, and immense knowledge, he always encouraged me to believe in my path and the value of my accomplishments. His excellent guidance aided me throughout my research and writing of this dissertation. I would like to thank my co-advisor, Professor Takáč for his outstanding help in all steps of my research. I cannot thank you enough for all your advice, encouragement, and numerous hours in discussions, which helped me develop my research ideas and get a deeper knowledge of my field. It has been very heart-warming to know there is someone to help, whenever you get into a struggle. I could not have envisioned having better advisors and mentors for my Ph.D. education.

I am also genuinely grateful to my committee members for all your invaluable feedback and helpful discussions. Professor Stolyar, thank you for all the kindness and patience in enlightening me on the first glance of research and building a profound background. Professor Scheinberg, thank you for all you did throughout my Ph.D. Your continual support and thought-provoking course shaped my academic path by solidifying my interest in Machine Learning. Dr. Kabul, thank you for providing me an opportunity to join your team as an intern and have access to the research facilities to incorporate my ideas in a real-world scale.

Outside of my dissertation committee, I was also greatly influenced and supported by other faculty members of Lehigh University. Professor Curtis, thank you for your insightful courses and all your support in whatever ways you could. Professor Storer and Wilson, I had the honor to be your teaching assistants and learn a lot from you. And, thank you Professor Napier and Yukich for your excellent courses.

I would like to thank my family for all their love and encouragement. For my parents who raised me with a desire of learning and my in-laws who encouraged me in all my pursuits. And most of all for my loving, supportive, encouraging, and patient wife Bita whose faithful support during all stages of my Ph.D. was unbelievable and admirable. Thank you a lot.

Contents

List of Tables

List of Figures

Abstract

Designing hand-engineered solutions for decision-making in complex environments is a challenging task. This dissertation investigates the possibility of having autonomous decision-makers in several real-world problems, e.g., in dynamic matching, marketing, and transportation. Achieving high-quality performance in these systems is strongly tied to the actions that a controller performs in different situations. This problem is further complicated by the fact that every single action might have long-term consequences, so ignoring them might cause unpredicted outcomes. My primary focus is to approach these problems with *long-term* objectives in mind, instead of only focusing on myopic ones. By borrowing techniques from optimal control and reinforcement learning, I design modeling infrastructures for each specific problem. Currently, the mainstream of reinforcement learning research uses games and robotics simulators for verification of the performance of an algorithm. In contrast, my main endeavor in this dissertation is to bridge the gap between the developed methods and their real-world applications, which are studied less often. For instance, for dynamic matching, I propose a simple matching rule with optimality guarantees; for customer journey, I use reinforcement learning to design an online algorithm based on temporal difference learning; and, for transportation, I showed that it is possible to train a solver with the capability of solving a wide variety of vehicle routing problems using reinforcement learning. Finally, I conclude this dissertation by introducing a new paradigm, which I call "corrective reinforcement learning." This paradigm addressed one major challenge in applying policies found by RL, that is, they might significantly differ from real systems. I propose a mechanism that resolves this issue by finding improved controllers which are close to the status quo. I believe that the models proposed in this dissertation will contribute to the

discovery of methods that can outperform current systems, which are primarily controlled by humans.

Chapter 1

Introduction

We begin this dissertation in Chapter 2 with a general dynamic matching problem and design a simple decision scheme that solves the long-term objective. We consider a matching system with random arrivals of items of different types. The items wait in queues—one per each item type—until they are “matched.” Each matching requires certain quantities of items of different types; after a matching is activated, the associated items leave the system. There exists a finite set of possible matchings, each producing a certain amount of “reward.” This model has a broad range of important applications, including assemble-to-order systems, Internet advertising, matching web portals, etc. We propose an optimal matching scheme in the sense that it asymptotically maximizes the long-term average matching reward, while keeping the queues stable. The scheme makes matching decisions in a specially constructed virtual system, which in turn control decisions in the physical system. The key feature of the virtual system is that, unlike the physical one, it allows the queues to become negative. The matchings in the virtual system are controlled by an extended version of the *greedy primal-dual* (GPD) algorithm, which we prove to be asymptotically optimal—this in turn implies the asymptotic optimality of the entire scheme. The scheme is real-time: at any time it uses simple rules based on the current state of the virtual and physical queues. It is very robust in that it does not require any knowledge of the item arrival rates, and automatically adapts to changing rates. The extended GPD algorithm and its asymptotic optimality apply to a quite general queueing network framework, not limited to matching problems, and therefore is of independent interest. This work is now published in Queueing

Systems [64].

We may find simple control policies with theoretical guarantees for problems like the matching problem of Chapter 2, but once the state space or the action space to represent the problems becomes large, these methodologies become intractable. One approach for resolving these limitations is to use *Reinforcement Learning* (RL), which is concerned with learning policies to maximize the cumulative reward. In the rest of this dissertation, we utilize the recent stream of developments in combining *deep learning* with RL, which is usually referred to as *Deep Reinforcement Learning* (DRL), for designing complicated policies.

Currently, the main research focus areas in AI are in specific test-beds: playing games, controlling toy robots, text translation, visual recognition, and many others. In contrast, my main goal is to bridge the gap between the developed methods and real-world applications, which is studied less often. In the next chapters, we specifically focus on RL and demonstrate how it can provide modeling infrastructures capable of finding effective policies.

In Chapter 3, we study the applicability of RL to a marketing problem in which a firm initiates different communications to its customers through marketing channels. Since we consider long-term interactions with customer, we refer to our specific marketing problem as *customer journey optimization*. We formulate the customer journey optimization problem within an RL framework, in which we try to maximize the *lifetime value* of customers instead of only considering myopic objectives. Using RL for the customer journey problem is still in its early stages, and most of the literature pursues an offline approach that lacks the adaptability to volatile customer behaviors. A major issue of using offline algorithms in real-world applications is that it is not possible to propagate interaction trajectories through time in order to learn a policy. In Chapter 3, we propose *Deep Concurrent TD* (DCTD), an extension of Concurrent TD [78] with neural networks, as the online algorithm that learns to maximize the long-term reward while keeping track of the customer reactions and adapting to their behavioral alterations. The strength of the DCTD algorithm is that it learns satisfactory policies only by observing the experiences of other customers, without knowing the environment dynamics. The basic idea is to learn a policy by exploration in a subset of the customers and then apply the successful ones to others. We develop a method from historical data to the online learner algorithm in which, before running a live policy, we train a reasonable one using an offline algorithm, e.g., DQN [61], as a warm-start policy. Experimentally, we show how DQN's performance degrades in exposure to varying customer behaviors; however, DCTD automatically adapts the policy according to non-stationary behaviors of the customers without knowing any a priori information about the changes. This framework can be employed in various domains, ranging from business applications to manufacturing as long as we have access to many concurrent environments.

In Chapter 4, we present an end-to-end framework for solving the Vehicle Routing Problem (VRP) using reinforcement learning. The VRP is a difficult combinatorial optimization problem with many exact and heuristic algorithms, but finding satisfactory results is still a hard task. In the simplest form, a single vehicle is responsible for delivering items to customers in the shortest set of routes. Motivated by the recent work by Bello et al. [8], we develop a framework for VRP with the capability of solving a wide variety of combinatorial optimization problems using RL. In this framework, we consider a parameterized stochastic policy, and by applying a policy gradient algorithm to optimize its parameters, the trained model produces the solution as a sequence of consecutive actions in real time, without the need to re-train for every new problem instance. One can consider the trained policy as a black-box (or meta-algorithm) heuristic with the capability of generating close-to-optimal tours in a reasonable amount of time. According to the findings of this chapter, our RL algorithm is competitive with state-of-the-art VRP heuristics both in solution quality and runtime, and this is progress toward solving the VRP with RL for real applications. Moreover, we show that our proposed framework can be applied to other variants of the VRP such as split-delivery VRP and stochastic VRP.

Although reinforcement learning can provide reliable solutions in many settings, practitioners are often wary of the discrepancies between the RL solution and their status quo procedures. Therefore, they may be reluctant to adapt to the novel way of executing tasks proposed by RL. On the other hand, many real-world problems require relatively small adjustments from the status quo policies to achieve improved performance. Therefore, in Chapter 5, we propose a student-teacher RL mechanism in which the RL (the "student") learns to maximize its reward, subject to a constraint that bounds the difference between the RL policy and the "teacher" policy. The teacher can be another RL policy (e.g., trained under a slightly different setting), the status quo policy, or any other exogenous policy. We formulate this problem using a stochastic optimization model and solve it using a primal-dual policy gradient algorithm. We prove that the policy is asymptotically optimal. However, a naive implementation suffers from high variance and convergence to a stochastic optimal policy. With a few practical adjustments to address these issues, our numerical experiments confirm the effectiveness of our proposed method in multiple GridWorld scenarios.

Chapter 2

Reward Maximization in General Dynamic Matching Systems

2.1 Introduction

We consider a dynamic matching system with random arrivals. Items of different types arrive in the system according to a stochastic process and wait in their dedicated queues to be "matched." Each matching requires certain quantities of items of different types; after a matching is activated, the associated items leave the system. There exists a finite number of possible matchings, each producing a certain amount of "reward." The objective is to maximize long-term average rewards, subject to the constraint that the queues of currently unmatched items remain stochastically stable. In this chapter, we propose a dynamic matching scheme and prove its asymptotic optimality. (In fact, the policy works for a more general objective, being a concave function of the long-term rates at which different matchings are used.)

Figure 2.1 shows an example of a matching system with 4 item types. The items arrive as a random process, as individual items or in batches. The average arrival rate of type i items is α_i. There exist 3 possible matchings; e.g., $\langle 1, 2\rangle$ is a matching which matches one item of type 1 with one item of type 2. $\langle 2, 3, 4\rangle$ is another matching which matches one item of types 2, 3 and 4. (In general, unlike in this example, a matching may require more than one item of any given type.) A matching can only be applied if all contributing items are present in the system; and if it is applied, the contributing items instantaneously leave the system.

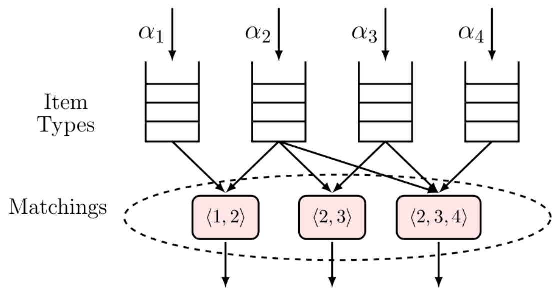


Figure 2.1 An example of the matching model.

The analysis of *static* matching has a large literature (see, e.g., [56]). The *dynamic* model, which we focus on, has attracted a lot of attention recently, due to a large variety of new (or relatively new) important applications. One example is assemble-to-order systems (see e.g., [71] and references therein), where randomly arriving product orders are "matched" with sets of parts required for the product assembly. Another application is Internet advertising [59], where the problem is to find appropriate matchings between the ad slots and the advertisers. Web portals as places for business and personal interactions are important applications; the problem in these portals (such as dating websites, employment portals, online games) is to match people with similar interests [13]. Matching problems also arise in systems with random arrivals of customers and servers; for example, in taxi allocation, where matched "items" are passengers and taxis [44]. Further applications also can be found in [14, 17].

Different control objectives may be of interest for matching systems. Gurvich and Ward [39] study the problem of minimizing finite-horizon cumulative holding costs for a model very close to ours. Plambeck and Ward [71], in the context of assemble-to-order systems, consider a model where item arrival rates can be controlled via a pricing mechanism; the objective includes queueing holding costs in addition to rewards/costs associated with order fulfillments, parts salvaging and/or expediting. Paper [71], in particular, proposes and studies a discrete-review policy; it involves solving an optimization problem at each review point.

A special case of the matching system, which received considerable attention, is where customers and servers randomly arrive to the system and each server can be matched with one customer from a certain subset. This model, also known as the (stochastic) bipartite matching system, was initially studied by [17]. The majority of the previous research for this model was focused on finding the stationary distribution [4, 3] and stability issues [13, 15, 58]. Bušić et al. [15] established necessary and sufficient conditions for stabilizability of such systems, and have shown that the well known MaxWeight algorithm achieves maximum stability region. The problem of minimizing the long-term average holding cost for the bipartite matching system is studied by [14]. They have shown that with known arrival rates (and some other conditions on the problem structure), a threshold-type policy is asymptotically optimal in the (appropriately defined) heavy traffic regime.

In this chapter, we show that the reward-maximizing optimal control of the matching model can be obtained by putting it into a typical queueing network framework. Our scheme uses a specially constructed virtual system, whose state, along with the state of the physical system, determines control decisions via a simple rule. In the virtual system any matching can be applied at any time and the queues are allowed to be negative. The matchings in the virtual system are controlled by (an extended version of) the Greedy Primal-Dual (GPD) algorithm [82], which maximizes a queueing network utility subject to stability of the queues. Negative queues in the virtual system can be interpreted as shortages of physical items of the corresponding types. The GPD algorithm in [82] does not allow negative queues, so it is insufficient for the control of our virtual system. *The main theoretical contribution* of this chapter is that we introduce and study an *extended version of GPD,* labeled EGPD, which *does allow negative queues,* and prove its asymptotic optimality under non-restrictive conditions that we specify. The approach of using a virtual system to control the original one has been used before, e.g., in [84], but the virtual system employed in this chapter is substantially different, primarily because it allows negative queues.

Our proposed scheme is very robust in that it does not require a priori knowledge of item arrival rates, and automatically adjusts if/when the arrival rates change. It also covers a wide range of applications and control objectives. For example, in the context of assemble-to-order systems, the objective can include rewards/costs associated with order fulfillments, parts salvaging and/or expediting.

Although our scheme is designed (and proved asymptotically optimal) for the reward maximization objective, which does *not* include holding costs, we will discuss heuristic approaches to how the scheme can be used to achieve good performance in terms of a more general objective (including holding costs).

The chapter is organized as follows. Section 2.2 contains notation used throughout the chapter. In Section 2.3 we formally introduce the matching model and the reward maximization problem; here we also formally define the corresponding virtual system and the overall control scheme, in which the matching algorithm for the virtual system is a key part. In Section 2.4, we introduce the Extended Greedy Primal-Dual (EGPD) algorithm for a general network model, with queues that may be negative, and prove asymptotic optimality of EGPD; here we also show that the virtual system algorithm (in Section 2.3) is a special case of EGPD and thus is asymptotically optimal. (A reader interested mostly in applications of our proposed scheme may skip Section 2.4, at least at first reading.) We evaluate the performance of our scheme via simulations in Section 2.5. Finally, in Section 2.6, we discuss heuristics in which a more general objective, including holding costs, can be addressed by tuning EGPD parameters. Some conclusions are given in Section 2.7.

2.2 Basic Notation

We denote by $\mathbb{R}$, $\mathbb{R}_+$ and $\mathbb{R}_-$ the set of real, real non-negative and real non-positive numbers, respectively. $\mathbb{R}^N$, $\mathbb{R}^N_+$ and $\mathbb{R}^N_-$ are the corresponding N-dimensional vector spaces. A vector $x \in \mathbb{R}^N$ is often written as $x = (x_n, n \in \mathcal{N})$, where $\mathcal{N} = \{1, 2, \cdots, N\}$. For two vectors $x, y \in \mathbb{R}^N$, $x \cdot y = \sum_{n=1}^N x_n y_n$ is the scalar (dot) product; vector inequality $x \leq y$ is understood component-wise. The standard Euclidean norm of x is denoted by $\|x\| = \sqrt{x \cdot x}$. The distance between point x and set $V \subseteq \mathbb{R}^N$ is denoted by $\rho(x, V) = \inf_{y \in V} \|x - y\|$.

For a vector function $f : \mathbb{R}_+ \to \mathbb{R}^N$ and a set $V \subseteq \mathbb{R}^N$, the convergence $f(t) \to V$ means that $\rho(f(t), V) \to 0$ as $t \to \infty$.

For differentiable functions $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R}^N \to \mathbb{R}$, we use $f'(t)$ (or $(d/dt)f(t)$) to denote the derivative with respect to t and $\nabla g(x) = ((\partial/\partial x_n)g(x), n \in \mathcal{N})$ is the gradient of g at $x \in \mathbb{R}^N$.

For a set V and a real-valued function $g(v)$, $v \in V$,

$$\arg\max_{v \in V} g(v)$$

denotes the subset of vectors $v \in V$ that maximize $g(v)$.

For $\xi, \eta \in \mathbb{R}$ and $\gamma \in \mathbb{R}_+$, we denote: $\xi \wedge \eta = \min\{\xi, \eta\}$, $\xi \vee \eta = \max\{\xi, \eta\}$; $\xi^+ = \xi \vee 0$, $\xi^- = (-\xi) \vee 0$; $[\xi]^+_\gamma = \xi$ if $\gamma > 0$ and $[\xi]^+_\gamma = \max\{\xi, 0\}$ if $\gamma = 0$.

Abbreviation *a.e.* means *almost everywhere* with respect to Lebesgue measure.

2.3 Optimal Control of the Matching System

The outline of this section is as follows. First, we formally define the *physical* matching system in Section 2.3.1 and discuss the flexibility of this model to include a large variety of practical systems in Section 2.3.2. In Section 2.3.3 we introduce a virtual system, corresponding to the physical one. In Section 2.3.4 we define a control scheme, such that a certain algorithm runs on the virtual system, and control decisions for the physical system depend on those in the virtual one. We propose a specific algorithm for the virtual system in Section 2.3.5; this algorithm is asymptotically optimal in the sense that, under certain non-restrictive conditions, when the algorithm parameter (β) goes to zero, our entire physical/virtual control scheme maximizes average matching reward in the physical system. (The asymptotic optimality will be proved later, in Section 2.4.) We discuss features of the virtual system algorithm, and the conditions for its asymptotic optimality in Section 2.3.6.

2.3.1 Definition of the Physical Matching System

Consider a matching system with I *item types* forming set $\mathcal{I} = \{1, \cdots, I\}$. The items arrive in *batches*, consisting of items of the same or different types. To simplify exposition, assume that batches arrive according to a Poisson process, with each batch type chosen upon arrival, independently, according to some fixed distribution. There is a finite number of possible batch types. The average rate at which type i items arrive into the system is $\alpha_i > 0$.

There is a finite set $\mathcal{J} = \{1, \cdots, J\}$ of possible *matchings.* Let $\mu(j) = (\mu_i(j),\, i \in \mathcal{I})$, where $\mu_i(j) \geq 0$ is the required number of type i items to form matching $j \in \mathcal{J}$. Without loss of generality, we can and do assume that the "empty" matching, with all $\mu_i = 0$, is an element of $\mathcal{J}$; the empty matching is denoted $\langle \emptyset \rangle$. If a matching requires either zero or one item of each type, it is denoted by the subset of the required item types; say, $\langle 1, 2\rangle$ denotes the matching requiring one item of type 1, one item of type 2, and zero of all other types.

Without loss of generality, we can and do assume that the matching decisions are made only at the times of batch arrivals into the system. Essentially without loss of generality, we also assume that at those times at most $m \geq 1$ matchings can be done. To simplify exposition, we further assume that $m = 1$—it will be clear from our analysis that all results and (with very minor adjustments) proofs hold for arbitrary fixed m. Therefore, from now on we consider the system as operating in discrete (slotted) time $t = 0, 1, 2, \ldots$, with i.i.d. batches arriving at those times, and exactly one (possibly empty) matching activated at each t.

Further, without loss of generality, we adopt the convention that the items that arrive at time t are only available for matching at time $t + 1$. (If items arriving at time t are immediately available for matching, the convention still holds if we simply pretend that they arrived at time $t - 1$, after the matching decision at time $t - 1$ was made.)

Type $i \in \mathcal{I}$ items waiting to be matched form a first-come,first-served (FCFS) queue; its length is denoted $\hat{Q}_i$. At any time t, any one matching $j \in \mathcal{J}$ can be activated subject to the constraint that all the required items must be available in the system. With activation of matching $j \in \mathcal{J}$,

(i) certain (real-valued) reward w_j is generated;

(ii) number $\mu_i(j)$ of items is removed from the queues of the corresponding types i.

Let X_j be the long-term average reward generated by matching j, under a given control policy. We are interested in finding a dynamic matching policy, which maximizes a continuously differentiable concave utility function $G(X_1, \cdots, X_J)$ subject to the constraint that all queue lengths $\hat{Q}_i(\cdot)$ remain stochastically stable. Informally speaking, stochastic stability means that as time goes to infinity the queues do not "run away" to infinity, i.e., remain $O(1)$. Formally, by stochastic stability we will understand positive recurrence of the underlying Markov process, describing the system evolution. (For example, if the process is a countable-state-space irreducible Markov chain, positive recurrence is equivalent to the existence of unique stationary probability distribution and to ergodicity.) Therefore, stochastic stability ensures that all arriving items are matched, without the backlogs and waiting times of unmatched items building up to infinity over time.

Remark 2.3.1. ***Stability and long-term averages.*** *We will give a specific definition of long-term average rewards X_j later. When the process is Markov and positive recurrent, then X_j can be thought of as the* steady-state average reward *u_j due to type j matchings—we will elaborate on the relation between X_j and u_j later.*

Remark 2.3.2. ***More general $\mu_i(j)$.*** *Our model and the results hold—as is—in the case in which the values of $\mu_i(j)$ can be real numbers of any sign. A negative $\mu_i(j)$ means that matching j* adds *$|\mu_i(j)|$ items to $\hat{Q}_i$, and by convention any negative number of items of any type is always available for matching completion. We assume in this chapter that $\mu_i(j)$ are non-negative integers to keep the exposition intuitive.*

2.3.2 Model Flexibility

The matching model defined in Section 2.3.1 is flexible enough to include a variety of systems and their features. Let us consider assemble-to-order systems as an example. In such systems, orders for multiple products arrive as a random process. Each product requires a certain number of components of each type to be assembled. Components also arrive into the system as a random process. A product can only be assembled when all necessary parts are available; in which case it brings a certain reward (profit). This is a matching system where the components and product-orders of different types are "items", a completed product is a matching comprising one corresponding product-order and the required number of parts. Salvaging and/or disposing of the components is easily accommodated; namely, salvaging/disposing of one component, labeled as a type i item, can be treated as a matching

$\langle i \rangle$, with a reward that might be negative (as well as non-negative). Similarly with orders: discarding an order for a product, which is labeled as item type ℓ, is a matching $\langle \ell \rangle$ with the corresponding (most likely, negative) reward. Expediting component delivery can be included as well. Suppose matching $\langle 1, 2, 3, 9 \rangle$ corresponds to product 9 assembled from (one unit of) parts $1, 2, 3$, with a reward of 20. However, the system has an option of expediting component 2, and receive it immediately, at a cost of 15. Then, assembling product 9 from already available components 1 and 3, and expedited component 2, can be modeled as a matching $\langle 1, 3, 9 \rangle$ with reward $20 - 15 = 5$. (Another, more natural, way to model expediting of item 2 is to treat it as a "matching," requiring -1 type-2 items, with a reward of -15. See Remark 2.3.2 above.)

This discussion illustrates the flexibility of our model *as long as the objective is to maximize average rewards associated with actions*, such as matching, salvaging, expediting, etc. The model does *not* explicitly include holding costs. In Section 2.6 we propose and discuss heuristic extensions of our scheme which do implicitly take holding costs into account.

2.3.3 Virtual Matching System

We will propose a matching control scheme in Section 2.3.4, which in parallel to the physical system "runs" a virtual system, which determines the matching decisions for the physical one. The virtual matching system is defined as follows.

The virtual system has the same item types, set of matchings and arrival flows as the physical system. It is only different in that any matching can be activated at any time and the queues of the virtual system can be negative, as well as positive. The matchings in the virtual system are activated based on its own state, regardless of the state of physical system. The activated matchings in the virtual system become actual matchings in the physical system either immediately, or later in time, depending on the availability of physical items. The virtual matchings, until they become actual ones, are called *incomplete* matchings. Incomplete matchings wait in a queue, which lists the incomplete matchings (their identities j) in the order of arrival; we denote the length of this queue by $\hat{Q}_0$. An incomplete matching becomes an actual one and leaves this queue when it is "completed" by all required physical items. (The queue of incomplete matchings, as we will see shortly, serves as the "interface" between the virtual and physical systems. In our figures and plots it is shown as part of the physical system.)

2.3.4 Control of the Physical Matching System via the Virtual System

Denote by $Q(t) = (Q_i(t), i \in \mathcal{I})$ and $\hat{Q}(t) = (Q_i(t), i \in \mathcal{I})$ the vectors of queue lengths in the virtual and physical systems, respectively, at time t. In this chapter, we always assume that the system is initialized in a state such that all physical and virtual queues are zero, $Q_i(0) = \hat{Q}_i(0) = 0,\ \forall i \in \mathcal{I}$, and there are no incomplete matchings, $\hat{Q}_0(0) = 0$. This means that the only feasible system states are those reachable from this "zero-state."

At time t the following occurs sequentially:

(i) A new matching is chosen in the virtual system based on $Q(t)$. (We will give a specific rule in Section 2.3.5.) If it is a non-empty matching j, then the virtual queues are updated as $Q := Q - \mu(j)$, and a new type-j incomplete matching is created and placed at the end of the (incomplete matchings) queue; so that $\hat{Q}_0 := \hat{Q}_0 + 1$.

(ii) The incomplete matchings queue is scanned in FCFS order, to find the first incomplete matching j', which can be completed, i.e., such that $\hat{Q}(t) \geq \mu(j')$. If such a matching j' is found, it is completed, i.e., it is removed from the incomplete matchings' queue (so that $\hat{Q}_0 := \hat{Q}_0 - 1$), a physical matching j' is created, and the corresponding number of physical items leaves the system, $\hat{Q} := \hat{Q} - \mu(j')$.

(iii) Both Q and $\hat{Q}$ are increased as: $Q := Q + \lambda(t)$, $\hat{Q} := \hat{Q} + \lambda(t)$; here $\lambda(t) = (\lambda_i(t), i \in \mathcal{I})$ is the random vector of arrivals of different types at t.

According to steps *(i)*-*(iii)* above, if matching $j \in \mathcal{J}$ is chosen in the virtual system at time t, the virtual queues change as follows:

$$Q(t+1) = Q(t) + \lambda(t) - \mu(j). \tag{2.1}$$

The evolution of the physical queues, if matching $j' \in \mathcal{J}$ is completed, is:

$$\hat{Q}(t+1) = \hat{Q}(t) + \lambda(t) - \mu(j')$$

Recall that we only consider feasible states of the queues—those reachable from the state where all virtual and physical queues are zero. Then we can make the following observations for the control scheme described above. For illustration, we will use Figure 2.2 showing a physical matching system with two item types and one possible matching and its corresponding virtual system. The system state shown in Figure 2.2 is such that: (a) in the physical system there are two type-1 items and no type-2 items; (b) the queue lengths in the virtual system are $Q_1(t) = 1$, $Q_2(t) = -1$; (c) there is one incomplete matching $\langle 1, 2\rangle$, which is incomplete because, while there is a type-1 item in the physical system (to complete it), there is no available (physical) type-2 item. (Note that at this point we did not specify yet the matching rule(s) for the virtual system—this will be done in Section 2.3.5. So, the state in Figure 2.2 only illustrates the relation between virtual and physical systems, not a specific matching rule.)

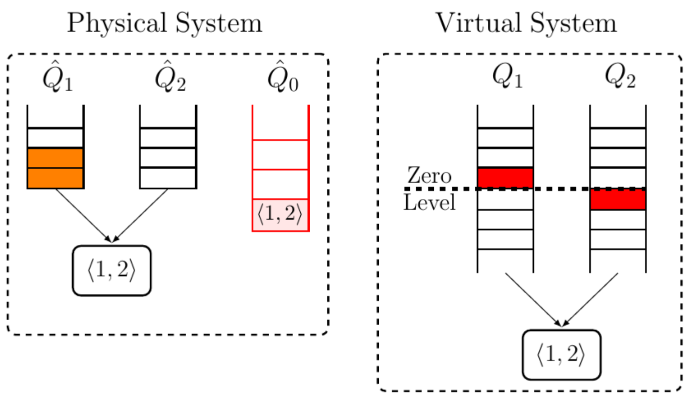


Figure 2.2 An example of the physical and virtual matching systems.

In a general system, if $Q_i(t) < 0$, then $Q_i^-(t) = |Q_i(t)|$ is the current shortage of type-i items for completing all incomplete matchings. (In Figure 2.2, $Q_2 = -1$ indicates the shortage of one type-2 item for completion of the incomplete matching $\langle 1, 2\rangle$.) If $Q_i(t) \geq 0$, then $Q_i^+(t) = Q_i(t)$ is the current surplus of type-i items, beyond what is needed for completing all incomplete matchings. (In Figure 2.2, $Q_1 = 1$ indicates that there is one type-1 item in addition to one type-1 item which can be used for completion of the incomplete matching $\langle 1, 2\rangle$.)

In addition to the notations $Q(t)$ and $\hat{Q}(t)$, let us denote by $\hat{\mathcal{Q}}_0(t)$ the state (list) of all incomplete matchings at time t.

The following simple proposition gives a total queue length bound (2.2) for the physical system in terms of the virtual one. This bound does not require any additional assumptions. Statements (ii) and (iii) of the proposition involve the notion of stochastic stability, which means positive recurrence of a Markov process. To keep the exposition simple, assume that the process $(Q(t), t \geq 0)$, describing the evolution of the virtual system, and the process $[(Q(t), t \geq 0), (\hat{Q}(t), t \geq 0), (\hat{\mathcal{Q}}_0(t), t \geq 0)]$ describing the evolution of the entire system, are countable-state-space Markov chains. (This is the case, for example, under the virtual system matching algorithm that we propose below in Section 2.3.5, and under linear utility function G.)

Proposition 2.3.3. *(i) At any $t \geq 0$, the following relation between physical and virtual queues holds:*

$$\hat{Q}_0(t) \leq \sum_i Q_i^-(t), \quad \sum_i \hat{Q}_i(t) \leq \sum_i Q_i^+(t) + \mu^* \sum_i Q_i^-(t) \leq \mu^* \sum_i |Q_i(t)|, \tag{2.2}$$

where $\mu^* \doteq \max_j \sum_i \mu_i(j)$.
(ii) Stochastic stability of $(Q(t), t \geq 0)$ implies that of $[(Q(t), t \geq 0), (\hat{Q}(t), t \geq 0), (\hat{\mathcal{Q}}_0(t), t \geq 0)]$.
(iii) If $(Q(t), t \geq 0)$ is stochastically stable, then the steady-state average rates at which different matchings are activated are the same in the physical and virtual systems.

Proof. (i) Clearly, for all $i \in \mathcal{I}$ at all times, $Q_i(t) \leq \hat{Q}_i(t)$. Note that the total shortage of items of all types for the completion of all incomplete matchings is $\sum_i Q_i^-(t)$; this means, in particular, that the total number of incomplete matchings is upper bounded as $\hat{Q}_0(t) \leq \sum_i Q_i^-(t)$. The total number of physical items in the system, $\sum_i \hat{Q}_i(t)$, can be partitioned into those that are ready to be used for completion of incomplete matchings and the "surplus" items; the number of the former is upper bounded by $\mu^* \hat{Q}_0(t)$; the number of the latter is equal to $\sum_i Q_i^+(t)$. This implies the second part of (2.2).
(ii) Follows from (i).
(iii) Follows from (ii). □

Remark 2.3.4. If $m \geq 1$ matchings can be done after each arrival, the sequence of steps (i)-(iii) above is repeated m times.

2.3.5 Asymptotically Optimal Matching Algorithm for the Virtual System

We now specify the algorithm to be used for the control of the virtual system. This algorithm will be proved to be asymptotically optimal for the virtual system, and then (by Proposition 2.3.3) for the physical system as well—see Remark 2.3.6 below.

Algorithm 1 Matching Algorithm for the Virtual System.

Let a (small) parameter $\beta > 0$ be fixed. At each time $t = 1, 2, \cdots$, activate matching

$$j(t) \in \arg\max_{j \in \mathcal{J}} \left[(\partial G(X(t))/\partial X_j)\ w_j + \sum_{i \in \mathcal{I}} \beta Q_i(t)\ \mu_i(j) \right], \tag{2.3}$$

where running average values $X_j(t)$ (of the rewards obtained by activation of different matchings j) are updated as follows:

$$X_{j(t)}(t+1) = (1-\beta) X_{j(t)}(t) + \beta\, w_{j(t)}, \tag{2.4}$$

$$X_j(t+1) = (1-\beta) X_j(t), \qquad j \neq j(t), \tag{2.5}$$

and $Q_i(t)$ is updated according to rule (2.1) for all $i \in \mathcal{I}$.

Note that if the function G is linear, say $G(X) = \sum_j X_j$, then the partial derivatives in (2.3) are constant, and rule (2.3) becomes simply

$$j(t) \in \arg\max_{j \in \mathcal{J}} \left[w_j + \sum_{i \in \mathcal{I}} \beta Q_i(t)\ \mu_i(j) \right]. \tag{2.6}$$

Moreover, in this case the algorithm does *not* need to keep track of the averages $X_j(t)$. As a result, both processes $(Q(t), t \geq 0)$ and $[(Q(t), t \geq 0), (\hat{Q}(t), t \geq 0), (\hat{\mathcal{Q}}_0(t), t \geq 0)]$ are countable-state-space Markov chains.

Consider the following assumption on the model structure. It is stated informally—its precise meaning will be given (in a more general context) later in Assumption 2.4.2 (Section 2.4). Also, in Section 2.3.6 we explain why this assumption is non-restrictive.

Assumption 2.3.5. *For any subset $\bar{\mathcal{I}} \subseteq \mathcal{I}$, there exists a matching activation strategy under which the long-term average drift of queues $i \in \bar{\mathcal{I}}$ is strictly positive and the long-term average drift of queues $i \notin \bar{\mathcal{I}}$ is strictly negative.*

When parameter β is small, then the *running average $X_j(t)$ is (one notion of) a long-term average rate* at which rewards due to matching j are generated. (See Section 2.4.4.) We will prove in Section 2.4 (as a corollary of Theorem 2.4.4) that, *under Assumption 2.3.5, Algorithm 1 is asymptotically optimal* in the following sense. (It is described here informally—the formal result is Theorem 2.4.4, for the more general model in Section 2.4.) Let V be the set of those long-term rate vectors X that are achievable (by some control strategy) subject to the stability of the queues, and let V^* be its optimal subset, $V^* = \arg\max_{X \in V} G(X)$. Then, when β is small, $X(t) \to V^*$ as $t \to \infty$.

Suppose now that the system process is Markovian under Algorithm 1 (as is the case when the function G is linear). Then Assumption 2.3.5 ensures process stability (for example, by the argument described in Section 4.9 in [82]). In this case the steady-state average rewards (due to different matchings) $u = (u_1, \ldots, u_J)$ are well defined. If the process is in the stationary regime, then obviously $\mathbb{E}X(t) = u$. Furthermore, the asymptotic optimality of Algorithm 1, in the sense described above, can be used to show that, as $\beta \to 0$, the vector u converges to the optimal set V^* (see Section 4.9 in [82]).

Remark 2.3.6. If Algorithm 1 is asymptotically optimal for the virtual system, then under our scheme it is also asymptotically optimal for the physical system. Indeed, the physical and virtual systems have the same set V of achievable long-term rate vectors X (subject to the stability of the queues). This is because any X achievable in the virtual system is achievable in the physical system as well (by our scheme, for which we have Proposition 2.3.3), and vice versa because obviously any control of the physical system can be applied to the virtual system. Therefore, under our scheme, if the virtual system produces (in the asymptotic limit) the optimal long-term rates $X \in V^*$, the same optimal rates are produced (by Proposition 2.3.3) in the physical system.

2.3.6 Discussion of Algorithm 1

Basic intuition

The key feature of the virtual system is that it has an option of creating matchings "in advance," before all required physical items have arrived. These "advance" matchings are the ones we called incomplete. Virtual queues keep track of the items' availability: recall that if $Q_i < 0$, $|Q_i|$ is the shortage of type i items, and if $Q_i \geq 0$, it is the surplus of type i items.

The intuition behind Algorithm 1 is the same as for the GPD algorithm in [82] (and other related works—see, e.g., [83] and references therein), but our model is more general in that the queues may have any sign. For simplicity of discussion, suppose the objective function is linear, $G(X) = \sum_j X_j$, in which case equation (2.3) in Algorithm 1 reduces to (2.6). The rule "tries" to choose a matching j which brings large reward w_j, but at the same time it "encourages" the drift of the queues towards 0. Indeed, recall that activation of any matching can only decrease the virtual queues. This means that the rule "encourages" the use of matchings that decrease positive Q_i's as much as possible and decrease negative Q_i's as little as possible; in other words, the rule encourages matchings requiring items of which there is a large surplus, and discourages matchings requiring items of which there is already a large shortage—this guarantees stability of the queues. When parameter β is small, the virtual queues "stabilize around correct levels"—positive or negative—which allows rule (2.6) to make "correct" decisions maximizing the average rewards.

Assumption 2.3.5 is non-restrictive

We now describe two common cases in which Assumption 2.3.5 holds. These two cases cover a very large number of applications.

Case 1. Assumption 2.3.5 holds automatically in the special case in which, for each item type i, there exists at least one matching requiring only type i items (namely, with $\mu_i \geq 1$ and $\mu_\ell = 0$ for $\ell \neq i$). In this case it suffices to pick any parameter m (the number of matchings per batch arrival) which is greater than $\mu^* \doteq \max_j \sum_i \mu_i(j)$. This special case is very common for the following reason, which we illustrate using the simple model in Figure 2.2. If matching $\langle 1,2\rangle$ is the only one possible (besides the empty matching), the system is unbalanced when the arrival rates are unequal, $\alpha_1 \neq \alpha_2$, and cannot be stable. (If items arrive one-by-one, this particular system obviously cannot be stable even if $\alpha_1 = \alpha_2$. More generally, any system with one-by-one arrivals cannot be stable if its "matching graph" is bipartite, see [58].) This shows that many practical systems typically need the option of using "single" matchings $\langle i\rangle$ anyway (salvaging or discarding individual items), to ensure stability, and then Assumption 2.3.5 holds.

Case 2. This case is more subtle. Suppose a system can potentially be made stable without requiring single-type matchings. For example, consider the system in Figure 2.2 in which the arrivals occur only in pairs $(1,2)$. Suppose also that up to two matchings can be done upon each arrival ($m = 2$). On the face of it, Assumption 2.3.5 does not hold for this system. Indeed, the linear relation $Q_1(t) = Q_2(t)$ holds at all times and, therefore, it is impossible for Q_1 and Q_2 to have different average drifts, which is required under Assumption 2.3.5. However, consider the orthogonal change of coordinates, $\tilde{Q}_1 = Q_1 + Q_2, \tilde{Q}_2 = Q_1 - Q_2$, with $\lambda(\cdot)$ and $\mu(\cdot)$ transformed accordingly. Then, $\tilde{Q}_2(t) \equiv 0$, and the system can be considered as having only one queue $\tilde{Q}_1$. For the latter system, Assumption 2.3.5 does hold. Note that *the algorithm itself does not need to perform any change of coordinates—it remains as is.* This situation is generic: if there is an inherent linear dependence between the queues, Assumption 2.3.5 often holds for the system after an appropriate orthogonal change of coordinates. This is, in fact, the case for many bipartite matching systems (with items arriving in pairs), including the one we consider later in Section 2.6.2.

To summarize the discussion in this subsection, Assumption 2.3.5 is essentially the assumption that the system can be made stable, plus a very common condition that the queues "can be moved in any direction" *within the subspace of feasible queue states.*

2.4 A General Network Model and EGPD Algorithm

In this section we introduce the *Extended Greedy Primal-Dual* (EGPD) algorithm for a general network model, which includes the matching system as a special case. This algorithm is a generalization of the GPD algorithm of [82] in the sense that queues at some network nodes, we call them *free* nodes, are allowed to have any sign; as they evolve, these queues are "free" to change from positive to negative and vice versa. The model in [82] is such that queues at all nodes are constrained to be non-negative—in our model we call such nodes *constrained*. First, we will formally define the model and the underlying optimization problem in Sections 2.4.1-2.4.3. The optimization problem determines the best possible (under any control algorithm) long-term drifts of the queues, which maximize the network "utility" subject to the condition that queue-drifts are zero at free nodes and are non-positive at constrained nodes; the optimal solutions to this problem give the maximum possible network utility that can be achieved by any network control strategy subject to stability of the queues. We define the EGPD algorithm in Section 2.4.4. In Section 2.4.5, we show that, as the algorithm parameter β goes to 0, the "fluid scaled" version of the process converges to a random process with sample paths being what we define as EGPD-trajectories. In Section 2.4.6 we prove asymptotic optimality of the EGPD-algorithm, in the sense that EGPD-trajectories converge to the optimal set of the underlying optimization problem while keeping all queues uniformly bounded; in other words, the EGPD-algorithm maximizes the system utility subject to stability. Finally, in Section 2.4.7 we show that Algorithm 1 (Section 2.3.5) for the virtual system of Section 2.3.3 is a special case of EGPD.

2.4.1 The Model

Consider a network consisting of a finite set of nodes $\mathcal{N} = \{1, 2, \cdots, N\}$, $N \geq 1$. The nodes are of two different types: N_1 *constrained* nodes form the set $\mathcal{N}^c = \{1, 2, \cdots, N_1\}$ and $N_2 = N - N_1$ *free* nodes form $\mathcal{N}^f = \{N_1 + 1, N_1 + 2, \cdots, N\}$. Either $\mathcal{N}^c$ or $\mathcal{N}^f$ is allowed to be an empty set. There is a queue associated with each node, where we denote by $Q_n(t)$ the queue length of node $n \in \mathcal{N}$ at time t and we will denote $Q(t) = (Q_n(t), n \in \mathcal{N})$. The queue length of node $n \in \mathcal{N}^c$ is always *non-negative*, but node $n \in \mathcal{N}^f$ can have queue length of any sign.

The system operates in discrete time $t = 1, 2, \cdots$. (By convention, we identify an integer time t with unit time interval [t,t+1), which is usually referred to as time slot t.) A finite number of *controls* is available, where we denote by K the set of controls. Upon activation of control $k \in K$ at time t, the following occurs sequentially:

(i) A certain (non-random) real amount ("number") $\mu_n(k) \geq 0$ of items is removed from queue n and leaves the network. Queues in constrained nodes cannot go below zero; so if $Q_n(t) \leq \mu_n(k)$, the entire content of queue n is removed.

(ii) A random (bounded) real amount ("number") $\lambda_n(k,t) \geq 0$ of items enters each node $n \in \mathcal{N}$, where $\lambda(k,t) = (\lambda_n(k,t), n \in \mathcal{N})$ are i.i.d. in time, with generic random variable denoted $\lambda(k) = (\lambda_n(k), n \in \mathcal{N})$.

According to steps *(i)* and *(ii)*, the queue update rules for constrained and free nodes, given control k is chosen at time t, are as follows:

$$Q_n(t+1) = [Q_n(t) - \mu_n(k)] \vee 0 + \lambda_n(k,t), \quad n \in \mathcal{N}^c \tag{2.7}$$

$$Q_n(t+1) = Q_n(t) - \mu_n(k) + \lambda_n(k,t), \quad n \in \mathcal{N}^f. \tag{2.8}$$

2.4.2 System Rate Region

For each $k \in K$ and time t, consider the random vector $b(k,t) = (b_n(k,t), n \in \mathcal{N})$ equal in distribution to $\lambda(k) - \mu(k)$. Clearly, $b(k,t)$ is equal to the random vector of queue increments $Q(t+1) - Q(t)$ provided that control k is chosen at time t and assuming $Q_n(t) \geq \mu_n(k)$ for all $n \in \mathcal{N}^c$. We call components of $b(k,t)$ the *nominal increments* of queues upon control k at time t. Let $k(t)$ denote the control chosen at time t by a given control policy.

Informally speaking, the finite-dimensional convex compact rate region $V \subset \mathbb{R}^N$ is defined as the set of all possible long-term average values of $b(k(t),t)$, which can be induced by different control policies. A formal definition of the rate region is as follows.

For each $k \in K$, denote by $\bar{b}(k) = \mathbb{E}b(k,t)$ the drift of queue lengths upon control k (at any time t when control k is activated). For a fixed probability distribution $\phi = (\phi_k, k \in K)$ (with $\phi_k \geq 0$ and $\sum_{k \in K} \phi_k = 1$) consider the vector

$$v(\phi) = \sum_{k \in K} \phi_k \bar{b}(k). \tag{2.9}$$

If we interpret ϕ_k as the long-term average fraction of time slots when control k is chosen from the set of controls K, then $v(\phi)$ corresponds to the vector of long-term average drifts of $Q(t)$, assuming that the queues in the constrained nodes never hit zero. Then the system rate region V is defined as the set of all possible vectors $v(\phi)$ corresponding to all possible ϕ.

2.4.3 Underlying Optimization Problem

Consider an open convex set $\tilde{V} \subseteq \mathbb{R}^N$ such that $\tilde{V} \supseteq V$. Consider a concave continuously differentiable *utility* function $H : \tilde{V} \to \mathbb{R}$ and the following optimization problem:

$$\begin{aligned} \max_{v \in V} \quad & H(v) \\ s.t. \quad & v_n \in \mathbb{R}_-, \; \forall\, n \in \mathcal{N}^c \\ & v_n = 0, \quad \forall\, n \in \mathcal{N}^f. \end{aligned} \tag{2.10}$$

Assumption 2.4.1. *Optimization problem* (2.10) *is feasible, i.e.*

$$\{v \in V : v_n \in \mathbb{R}_-, \forall n \in \mathcal{N}^c \textit{ and } v_n = 0, \forall n \in \mathcal{N}^f\} \neq \varnothing. \tag{2.11}$$

If Assumption 2.4.1 holds, we denote by $V^* \subseteq V$ the set of optimal solutions of (2.10). The dual to optimization problem (2.10) is

$$\min_{(y_n \in \mathbb{R}_+, n \in \mathcal{N}^c), (y_n \in \mathbb{R}, n \in \mathcal{N}^f)} \left(\max_{v \in V} \left(H(v) - y \cdot v \right) \right), \tag{2.12}$$

and we denote by Q^* the closed convex set of optimal solutions $q^* \in \mathbb{R}_+^{N_1} \times \mathbb{R}^{N_2}$ of problem (2.12). For any $v^* \in V^*$ and any $q^* \in Q^*$, the compementary slackness condition holds:

$$q^* \cdot v^* = 0. \tag{2.13}$$

In Section 2.4.4, we will introduce an algorithm that is asymptotically optimal under the following assumption, which is stronger than Assumption 2.4.1.

Assumption 2.4.2. *For any subset $\bar{\mathcal{N}}^f \subseteq \mathcal{N}^f$, there exists $v \in V$ such that $v_n > 0$ for $n \in \bar{\mathcal{N}}^f$ and $v_n < 0$ for $n \notin \bar{\mathcal{N}}^f$.*

Assumption 2.4.2 means that there always exists a control policy which provides, simultaneously, a *strictly negative average drift* to all the constrained node queues and *non-zero average drifts* toward zero for all free node queues.

Note that under Assumption 2.4.2, the set Q^* is compact. Indeed, the optimal value of problem (2.10) is equal to

$$H(v^*) = \max_{v \in V} \left(H(v) - q^* \cdot v\right) \tag{2.14}$$

for any $v^* \in V^*$ and any $q^* \in Q^*$. The set Q^* must be bounded, because otherwise, from Assumption 2.4.2, there would exist $v \in V$ such that $v_n < 0$ for all nodes with $q_n \geq 0$, and $v_n > 0$ for all nodes with $q_n < 0$. Then we can arbitrarily increase the right hand side of (2.14) by choosing $q^* \in Q^*$ with large $|q_n^*|$.

The problem that we are going to address is as follows. Let X denote a long-term average value of $b(k(t), t)$ under a given dynamic control policy, that is, a policy of choosing $k(t)$ depending on the system state. We are interested in finding a dynamic control policy such that when optimization problem (2.10) is feasible, and moreover, the stronger Assumption 2.4.2 holds, the corresponding X is close to V^*, while the system queues remain stochastically stable.

2.4.4 Extended Greedy Primal-Dual Algorithm

Consider the control policy in algorithm 2.

Algorithm 2 EGPD algorithm for the general network model

At time $t = 1, 2, \cdots$, choose a control

$$k(t) \in \arg\max_{k \in K} [\nabla H(X(t)) - \beta Q(t)] \cdot \bar{b}(k), \tag{2.15}$$

where $\beta > 0$ is a small parameter. Here $X(t)$ is the running average of $b(k(t), t)$, updated as follows:

$$X(t+1) = (1-\beta)X(t) + \beta\, b(k(t), t) \tag{2.16}$$

and $Q(t)$ is updated according to (2.7) and (2.8).

The initial condition is $X(0) \in \tilde{V}$. Note that this initial condition and update rule (2.16) imply that $X(t) \in \tilde{V}$ for all $t \geq 0$. Hence the system evolution is well-defined for all $t \geq 0$, since the gradient and argmax in (2.15) are well-defined.

Also note that, if $0 < \beta < 1$, then for $t \geq 1$

$$X(t) = \sum_{\tau=0}^{t-1} \beta(1-\beta)^{t-1-\tau} b(k(\tau), \tau) + (1-\beta)^t X(0).$$

Therefore, when t is large, $X(t)$ is essentially the geometric average of values of $b(k(\tau), \tau)$ up to time $t-1$. When t is large and $\beta > 0$ is small, $X(t)$ is (one notion of) the long-term average of values of $b(k(\tau), \tau)$ up to time $t-1$.

2.4.5 Asymptotic Regime and Fluid Limit

We define the *EGPD-trajectory* as a pair of absolutely continuous functions (x, q)= $((x(t), t \geq 0), (q(t), t \geq 0))$, each taking values in $\mathbb{R}^N$ and satisfying the following conditions:

(i) For all $t \geq 0$,

$$x(t) \in \tilde{V} \tag{2.17}$$

and for almost all $t \geq 0$,

$$x'(t) = v(t) - x(t), \tag{2.18}$$

where

$$v(t) \in \arg\max_{v \in V} [\nabla H(x(t)) - q(t)] \cdot v. \tag{2.19}$$

(ii) We have

$$q_n(t) \geq 0, \forall t \geq 0, \ n \in \mathcal{N}^c \tag{2.20}$$

$$q_n'(t) = [v_n(t)]^+_{q_n(t)}, \quad a.e. \ in \ t \geq 0, \, n \in \mathcal{N}^c \tag{2.21}$$

$$q_n'(t) = v_n(t), \quad a.e. \ in \ t \geq 0, \, n \in \mathcal{N}^f \tag{2.22}$$

Functions $x(t)$ and $q(t)$ are dynamically changing primal and dual variables, respectively, for problems (2.10) and (2.12), which arise as asymptotic limits of the fluid scaled version of the process as described next.

Consider a sequence of processes (X^β, Q^β), indexed by a parameter β, where $\beta \downarrow 0$ along a sequence $\mathcal{B} = \{\beta_j\}_{j=1}^\infty$ with $\beta_j > 0$ for all j. The initial state $(X^\beta(0), Q^\beta(0)) \in \tilde{V}$ is fixed for each $\beta \in \mathcal{B}$. (The processes and variables associated with a fixed parameter β will be supplied by superscript β.)

We need to augment the definition of the process. Let us assume $X^\beta(t)$ and $Q^\beta(t)$ are functions defined on $t \in \mathbb{R}_+$ and constant within each time slot $[l, l+1)$, $l = 0, 1, 2, \cdots$. Thus for each β, consider the (continuous-time) process $Z^\beta = (X^\beta, Q^\beta)$, where

$$X^\beta = (X^\beta(t) = (X_n^\beta(t), n \in \mathcal{N}), t \geq 0), \tag{2.23}$$

$$Q^\beta = (Q^\beta(t) = (Q_n^\beta(t), n \in \mathcal{N}), t \geq 0). \tag{2.24}$$

For each β,

$$z^\beta = (x^\beta, q^\beta) \tag{2.25}$$

is the fluid scaled version of process Z^β, obtained by

$$x^\beta = X^\beta(t/\beta), \quad q^\beta = \beta Q^\beta(t/\beta). \tag{2.26}$$

The following theorem is a straightforward modification of Theorem 3 in [82], which we present without proof.

Theorem 2.4.3. *Consider a sequence of processes $\{z^\beta\}$ with $\beta \downarrow 0$ along set $\mathcal{B}$. Each process is considered as a random element in the Skorohod space of RCLL ("right continuous with left limits") functions. Assume that $z^\beta(0) \to z(0)$, where $z(0)$ is a fixed vector in $\mathbb{R}^{2N}$ such that $X(0) \in \tilde{V}$. Then, the sequence $\{z^\beta\}$ is relatively compact and any weak limit of this sequence (i.e a process obtained as the weak limit of a subsequence of $\{z^\beta\}$) is a process with sample paths z being EGPD-trajectories (with initial state $z(0)$) with probability 1.*

2.4.6 Global Attraction Result

The following theorem is the main result of this section; it shows the convergence of EGPD-trajectories to the saddle set $V^* \times Q^*$.

Theorem 2.4.4. *Under Assumption 2.4.2, the following hold:*

(i) For any EGPD-trajectory (x, q), as $t \to \infty$,

$$x(t) \to V^*, \tag{2.27}$$

$$q(t) \to q^*, \text{ for some } q^* \in Q^*. \tag{2.28}$$

(ii) Let some compact subsets $V^\square \subset \tilde{V}$ and $Q^\square \subset \mathbb{R}_+^{N_1} \times \mathbb{R}^{N_2}$ be fixed. Then, the convergence

$$(x(t), q(t)) \to V^* \times Q^*, \quad t \to \infty, \tag{2.29}$$

is uniform across all EGPD-trajectories with initial states $(x(0), q(0)) \in V^\square \times Q^\square$.

The proof of Theorem 2.4.4 is a generalization of that of Theorem 2 in [82]—all steps of the latter are extended to our more general setting. For this reason we will not give a complete proof of Theorem 2.4.4 in this chapter. Instead, we demonstrate the key points involved in the generalization, by proving in this section the convergence (2.27) for the special case when $x(0) \in V$ and $H(\cdot)$ is *strictly* concave.

Consider a fixed EGPD-trajectory (x, q). The property

$$\rho(x(t), V) \leq \rho(x(0), V)e^{-t},\ t \geq 0 \tag{2.30}$$

holds regardless of Assumptions 2.4.1 or 2.4.2 (cf. Lemma 20 in [82]). This shows that the entire trajectory $(x(t), t \geq 0)$ is contained within V. Then, this fact implies that $\sup_{t\geq 0} \|\nabla H(x(t))\| < \infty$.

A time point $t \geq 0$ is called "*regular*" if conditions (2.17)-(2.19) are satisfied and proper derivatives $x'(t)$, $q'(t)$ and $f'(t)$ exist. Almost all t are regular.

Let us introduce the following function:

$$F(v, y) = H(v) - \frac{1}{2}\sum_{n\in\mathcal{N}} y_n^2, \quad v \in \tilde{V}, \quad y_n \in \mathbb{R}_+ \text{ for } n \in \mathcal{N}^c, \quad y_n \in \mathbb{R} \text{ for } n \in \mathcal{N}^f.$$

Lemma 2.4.5. *The trajectory* $(q(t), t \geq 0)$ *is such that*

$$\sup_{t\geq 0} \|q(t)\| < \infty. \tag{2.31}$$

Proof. Within this proof, we say that a vector function (or scalar function) $\alpha(t), t \geq 0$, is *uniformly bounded* if $\sup_{t\geq 0} \|\alpha(t)\| < \infty$. By Assumption 2.4.2, the following holds for some fixed number $\delta > 0$. For any $t \geq 0$, there exists $\xi = (\xi_n,\, n \in \mathcal{N}) \in V$ such that for any n, $|\xi_n| \geq \delta$, $\xi_n > 0$ if $q_n < 0$, and $\xi_n < 0$ if $q_n \geq 0$. Then for any regular $t \geq 0$ (and a corresponding ξ) we have:

$$\begin{aligned}
\frac{d}{dt}F(x(t), q(t)) &= [\nabla H(x(t)) - q(t)] \cdot v(t) - \nabla H(x(t)) \cdot x(t) \\
&\geq [\nabla H(x(t)) - q(t)] \cdot \xi - \nabla H(x(t)) \cdot x(t) \\
&= -\sum_{n\in\mathcal{N}} \xi_n q_n(t) + \nabla H(x(t)) \cdot (\xi - x(t)) \\
&\geq \delta \sum_{n\in\mathcal{N}} |q_n(t)| + \nabla H(x(t)) \cdot (\xi - x(t))
\end{aligned} \tag{2.32}$$

Since $\nabla H(x(t))$ and $x(t)$ are uniformly bounded, so is the second term in (2.32). When $\|q(t)\|$ is large, the first term in (2.32) is large positive. We conclude that $(d/dt)F(x(t), q(t)) \geq$

$\epsilon_1 > 0$ as long as $\|q(t)\| \geq C_1 > 0$, for some fixed constants ϵ_1 and C_1. Since $H(x(t))$ is uniformly bounded, we can pick $C_2 > 0$ sufficiently large so that $F(x(t), q(t)) \leq -C_2$ implies $\|q(t)\| \geq C_1$ and then $(d/dt)F(x(t), q(t)) \geq \epsilon_1 > 0$. We then conclude that $\liminf_{t\to\infty} F(x(t), q(t)) \geq -C_2$ and, therefore, $\inf_{t\geq 0} F(x(t), q(t)) > -\infty$. The latter (along with the uniform boundedness of $H(x(t))$) implies that $q(t)$ is uniformly bounded. $\square$

Lemma 2.4.6. *For any EGPD-trajectory, at any regular time $t \geq 0$,*

$$\frac{d}{dt}F(x(t), q(t)) = \nabla H(x(t)) \cdot (v(t) - x(t)) - q(t) \cdot v(t) \tag{2.33}$$

and

$$v(t) \in \arg\max_{v \in V} \nabla H(x(t)) \cdot (v - x(t)) - q(t) \cdot v \tag{2.34}$$

Furthermore, if Assumption 2.4.1 *holds,*

$$\frac{d}{dt}F(x(t), q(t)) \geq \nabla H(x(t)) \cdot (v^* - x(t)) \geq H(v^*) - H(x(t)). \tag{2.35}$$

Proof. Noting $q_n'(t) = v_n(t)$ and $v_n^* = 0$, for any $n \in \mathcal{N}^f$, every step of the proof is analogous to that of Lemma 3 in [82]. $\square$

Select an arbitrary point $q^* \in Q^*$ and associate it with the following function

$$F^*(v, y) = H^*(v) - \frac{1}{2}\sum_{n\in\mathcal{N}} (y_n - q_n^*)^2, \quad v \in \tilde{V}, \quad \begin{array}{l} y_n \in \mathbb{R}_+ \text{ for } n \in \mathcal{N}^c, \\ y_n \in \mathbb{R} \text{ for } n \in \mathcal{N}^f, \end{array}$$

where

$$H^*(v) = H(v) - q^* \cdot v$$

is the Lagrangian of problem (2.10) with the dual variable equal to $q^* \in Q^*$. Having strictly concave $H(\cdot)$ implies that $H^*(\cdot)$ is also a strictly concave function and

$$v^* = \arg\max_{v\in V} H^*(v) \tag{2.36}$$

is the unique optimal solution. Note that $\nabla H^*(v) = \nabla H(v) - q^*$.

Lemma 2.4.7. *Consider $F^*(\cdot,\cdot)$ associated with an arbitrary $q^* \in Q^*$. Then for all (regular) $t \geq 0$,*

$$\frac{d}{dt}F^*(x(t),q(t)) \geq [\nabla H(x(t)) - q^*] \cdot (v(t) - x(t)) - (q(t) - q^*) \cdot v(t) \tag{2.37}$$

and

$$x(t) \in V \text{ implies } \frac{d}{dt}F^*(x(t),q(t)) \geq 0. \tag{2.38}$$

Proof. The proof is analogous to that of Lemma 5 in [82]. The only difference is the existence of free nodes, where we can easily validate this Lemma by using $q_n'(t) = v_n(t)$ and $v_n^* = 0$ for any $n \in \mathcal{N}^f$. □

Proof of Theorem 2.4.4. The convergence (2.27) follows from an inequality that we first derive. For any (regular) $t \geq 0$,

$$\begin{aligned}\frac{d}{dt}F^*(x(t),q(t)) \geq& (\nabla H(x(t)) - q^*) \cdot (v(t) - x(t)) - (q(t) - q^*) \cdot v(t)\\ =& (\nabla H(x(t)) - q^*) \cdot (v^* - x(t)) - (q(t) - q^*) \cdot v(t) +\\ & (\nabla H(x(t)) - q^*) \cdot (v(t) - v^*)\\ =& \nabla H^*(x(t)) \cdot (v^* - x(t)) - (q(t) - q^*) \cdot v^* +\\ & (\nabla H(x(t)) - q(t)) \cdot (v(t) - v^*) \qquad (2.39)\\ =& B_1(t) + B_2(t) + B_3(t), \qquad (2.40)\end{aligned}$$

where $B_i(t)$, $i \in \{1,2,3\}$ is the ith term in the right hand side of (2.39). Since $x(t) \in V$ and v^* maximizes $H^*(\cdot)$ over the compact set V, we have

$$B_1(t) \geq H^*(v^*) - H^*(x(t)) \geq 0. \tag{2.41}$$

Thus, for any $\epsilon_1 > 0$, there exists sufficiently small $\epsilon_2 > 0$ such that

$$B_1(t) \geq \epsilon_2 \text{ as long as } \|x(t) - v^*\| \geq \epsilon_1. \tag{2.42}$$

Moreover, using complementary slackness (2.13),

$$B_2(t) = -(q(t) - q^*) \cdot v^* = -q(t) \cdot v^* = -\sum_{n \in \mathcal{N}^c} q_n(t)\, v_n^* \geq 0, \tag{2.43}$$

and

$$B_3(t) = (\nabla H(x(t)) - q(t)) \cdot (v(t) - v^*) \geq 0, \tag{2.44}$$

because $v(t)$ maximizes $\nabla H(x(t)) - q(t)) \cdot v$ over all $v \in V$.

Now, $\|x(t) - v^*\|$ must converge to zero (which proves (2.27)). Indeed, suppose not. Then, there exists $\epsilon_1 > 0$ and a sequence $t_n,\ n = 1, 2, \ldots,\ t_n \uparrow \infty$, such that $\|x(t_n) - v^*\| \geq 2\epsilon_1$. Since $x(t)$ is Lipschitz continuous, this implies that for some $\delta > 0$,

$$\|x(t) - v^*\| \geq \epsilon_1, \quad t_n \leq t \leq t_n + \delta, \quad \forall n,$$

and then, by (2.42), for some $\epsilon_2 > 0$,

$$B_1(t) \geq \epsilon_2, \quad t_n \leq t \leq t_n + \delta, \quad \forall n.$$

This means that $\int_0^\infty (d/dt) F^*(x(t), q(t)) = \infty$ (recall that $B_2(t)$ and $B_3(t)$ are non-negative), and therefore $F^*(x(t), q(t)) \to \infty$. But, this is impossible, because, by the definition of function F^* and Lemma 2.4.5, $\sup_{t \geq 0} \|F^*(x(t), q(t))\| < \infty$. The contradiction proves (2.27). □

2.4.7 Mapping of the Virtual Matching System of Section 2.3.3 into EGPD Framework

Now we are in position to show that Algorithm 1 for the control of the virtual system in the original matching model in Section 2.3 is a special case of the EGPD Algorithm. The mapping of the virtual system of Section 2.3.3 into the more general model of Section 2.4.1 is as follows. Consider the following system, which we refer to as a modification of the virtual system. Suppose the item types $\mathcal{I}$ are modelled as free nodes and let the set of matchings $\mathcal{J}$ be the set of controls K. Let us add one *constrained* node per matching $j \in \mathcal{J}$. (These additional nodes are the *utility* nodes in the terminology of GPD algorithm [82].) From this point on, for convenience of the notation, we replace the set of indices of item types $\mathcal{I}$ with $\{J+1, \cdots, J+I\}$ and denote by $\mathcal{I}^c = \{1, \cdots, J\}$ the set of all constrained nodes. For the constrained nodes we adopt the convention that they never receive any inputs, i.e., $\lambda_j(t) \equiv 0,\ j \in \mathcal{I}^c$. We also fix a sufficiently large $c > 0$, so that $w_j - c < 0$ for all constrained nodes, and for each constrained node (or, matching) $j \in \mathcal{I}^c$ we set by convention $\mu_j(j) = c - w_j > 0$ and $\mu_i(j) = c > 0,\ i \in \mathcal{I}^c \backslash \{j\}$. These conventions about the constrained nodes guarantee that under any control strategy, their queues are automatically stable. In fact, for any $i \in \mathcal{I}^c$ and any initial value $Q_i(0)$, the queue length $Q_i(t)$ will decrease until it hits 0 within a finite time and then it will remain at 0. This allows us to assume, without loss of generality, that $Q_i(t) \equiv 0$ for all constrained nodes .

For a matching (or, constrained node) j, we have $b(j,t) = (\lambda_i(t) - \mu_i(j),\ i \in \mathcal{I}^c \cup \mathcal{I})$ and $\bar{b}(j) = \mathbb{E}b(j,t)$. Note that, for $i \in \mathcal{I}^c$, $\bar{b}_i(j) = w_j - c$ if $i = j$ and $\bar{b}_i(j) = -c$ otherwise.

If $j(t)$ is the matching chosen at t, then the compact rate region $V \subset \mathbb{R}^{J+I}$ is the set of all possible vectors

$$v = (v_1, \ldots, v_J, v_{J+1}, \ldots, v_{J+I})$$

being possible long-term average values of $\bar{b}(j(t))$ under different matching strategies (see formal definition in Section 2.4.2).

Finally, we define the utility function $H(v)$ as follows:

$$H(v_1, \ldots, v_J, v_{J+1}, \ldots, v_{J+I}) = G(v_1 + c, \ldots, v_J + c).$$

Given these conventions, it is easy to see that the problem of maximizing $G(X_1, \ldots, X_J)$ (subject to the stability of the queues) in the original matching system is equivalent to the problem of maximizing $H(X_1, \ldots, X_J, X_{J+1},\ \ldots,\ X_{J+I})$ (subject to the stability of the queues) in the modified system defined in this subsection. The latter system is a special case of the general system of Section 2.4.1. If we specialize Algorithm 2 to the modified system, and then rewrite it in terms of the original virtual system, we obtain Algorithm 1. Assumption 2.4.2, specialized to the modified system and expressed in terms of the original virtual system, gives the formal meaning of Assumption 2.3.5 (which is stated informally).

The mapping described in this section and the asymptotic optimality of the EGPD algorithm under Assumption 2.4.2 imply the asymptotic optimality of Algorithm 1 under Assumption 2.3.5.

2.5 Simulations

In this section, we evaluate the performance of the EGPD algorithm via simulations. Consider the system described in Section 2.1. We extend the set of possible matchings by including "single" matchings (see Section 2.3.6):

$$\{\langle\varnothing\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle, \langle 1,2\rangle, \langle 2,3\rangle, \langle 2,3,4\rangle\}\,.$$

The reward vector is $w = (0, -1, -1, 1, 2, 5, 4, 7)$, where its jth component corresponds to the jth element of the set of matchings. We consider a linear utility function, namely the sum of average rewards due to different matchings. The vector of arrival rates is $\alpha = (1.2, 1.5, 2, 0.8)$ per second.

For our linear utility function, the EGPD algorithm for the virtual system is given by rule (2.6).

A. Average reward maximization. We use parameter $\beta = 0.01$. Figure 2.3 shows the queue trajectories of the virtual and physical systems under the EGPD algorithm. All queues are initially empty. We observe that all queues are quickly "converging". Nearly all type-2 and type-4 items are matched right after they enter the system, while there exists a queue of around 100 items of types 1 and 3.

The rates at which matchings are activated under the EGPD algorithm are provided in Table 2.1, which shows that these rates are close to the optimal ones, obtained by solving the underlying optimization problem (which is a linear program in this case). Therefore, as expected, the algorithm yields near-optimal performance for small β. Note that solving the optimization problem requires the knowledge of arrival rates (as well as other system parameters), while our algorithm need not know arrival rates.

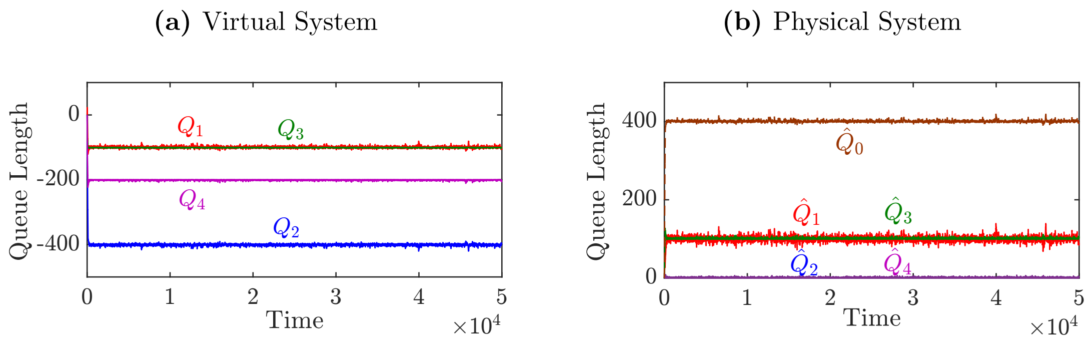


Figure 2.3 Queue trajectories of the virtual and physical systems under EGPD algorithm.

Table 2.1 Matching rates: Optimal vs. EGPD. (Runtime=30000 seconds)

Method	Matchings						
	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3\rangle$	$\langle 4\rangle$	$\langle 1,2\rangle$	$\langle 2,3\rangle$	$\langle 2,3,4\rangle$
EGPD	0	0	1.69345	0.4829	1.1924	0	0.31075
Optimal	0	0	1.70005	0.49995	1.2001	0	0.29975

Figure 2.4 demonstrates the average matching reward per unit time. We have calculated the optimal average reward (by solving the linear program) which is equal to 10.8, and plotted it on the figure. As is clear from the graph, the running average reward under the EGPD algorithm is getting very close to optimal objective value, and this convergence is sufficiently fast.

B. Effect of parameter β. In order for βQ in the virtual system to "stay close" to some $q^* \in Q^*$, the parameter β should be small. Therefore, *as long as the parameter β is sufficiently small*, the algorithm is nearly optimal and the virtual queue lengths are roughly of the order $1/\beta$. As β is increasing, the accuracy of the algorithm in terms of average reward maximization decreases, while the queues become smaller.

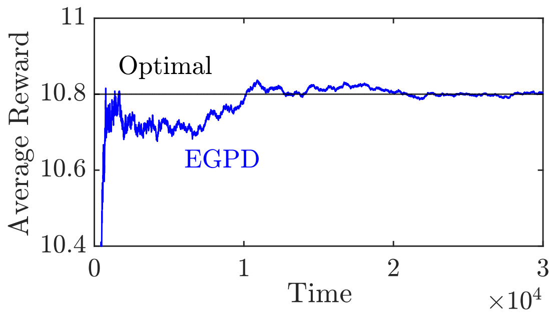


Figure 2.4 Average matching reward under the EGPD algorithm.

The dependence of the average reward on β for the considered scenario is shown in Figure 2.5. First, we note that the average reward remains nearly optimal for values of β almost as large as 1 (i.e., not even very small in absolute terms). Then, as β changes from 1 to about 10, the average reward decreases and reaches the lower "plateau," and then remains constant for $\beta \geq 10$. Thus, as expected, the algorithm is effective in terms of reward maximization when β is sufficiently small (less than 1 in our scenario); when β is sufficiently large (greater than 10 in our scenario), the average reward is also roughly independent of β, but is at a lower, suboptimal level.

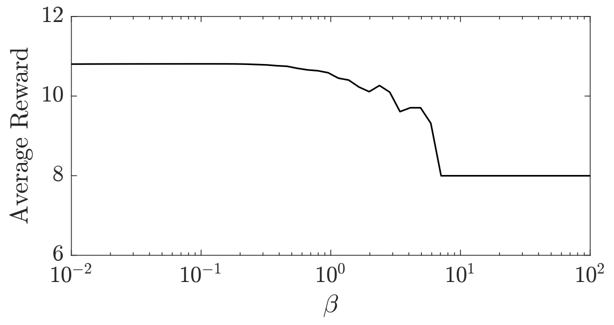


Figure 2.5 Average matching reward for different values of β.

We note that larger values of β have the benefit of reducing the queues and, as a result, reducing (as we will see next) the algorithm response (or, adaptation) time to changes of the items' arrival rates. (Shorter queues also mean lower holding costs, if such are a part of the model. This will be discussed in Section 2.6.) Therefore, the value of parameter β should be chosen, very informally speaking, "as large as possible, but not larger."

C. Automatic adaptation to changes in arrival process. An important robustness issue is how quickly the EGPD algorithm responds to changes in the arrival process. In the following experiment, the arrival rates are changed to $\alpha = (1.8, 0.8, 1.4, 1)$ at time 2000. This change leads to different optimal matching rates and thus different optimal values. If quick response to arrival rate changes is important, a larger β is preferable. Here we use $\beta = 0.1$. Figure 2.6 shows the queue trajectories of the virtual and physical systems. We observe that EGPD automatically adapts to the new arrival rates and reaches the new "right" queue lengths, without using any a priori information on this change.

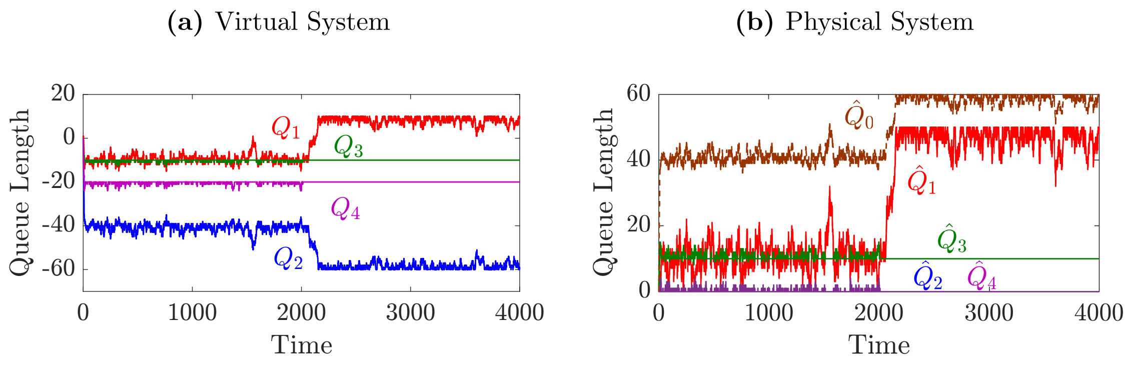


Figure 2.6 Adaptation to the changes in arrival rates.

2.6 Heuristics for the Objective Including both Matching Rewards and Holding Costs

2.6.1 General Discussion

The scheme we proposed in Section 2.3.4 for the matching model is asymptotically optimal for the reward maximization problem. (We will refer to this entire scheme as EGPD, because EGPD is its key part, applied to the virtual system and determining matching choices.) In practical systems, the objective may be more general, namely maximizing the average "profit" defined as average reward minus average queue holding cost. We now informally discuss how EGPD can be used to achieve better profit in the system (even though it is not specifically designed for that).

For the purposes of the discussion below, we assume linear holding costs with rate vector $c = (c_i,\ i \in \mathcal{I})$; that is the average holding cost over interval $[0, T]$ is

$$\frac{1}{T} \int_0^T c \cdot \hat{Q}(t) dt. \tag{2.45}$$

Suppose the arrival rates are scaled up by a factor $r > 0$. This simply speeds up the process r times, so that the average reward increases r times, while the holding cost remains same. Thus for systems with "high" arrival rates, the rewards dominate the profit objective and we expect the average profit obtained by the EGPD algorithm to be "close" to the optimal one. In other cases, holding costs may dominate, for example when the system is in (appropriately defined) heavy traffic (see [39, 14])—this makes the queues necessarily large. When the optimal average rewards and optimal holding cost are on the same scale, the EGPD parameter settings can be used to control the tradeoff between these two performance measures, thus potentially improving the average profit. We now briefly discuss different heuristic approaches for profit improvement within the framework of our scheme.

Choice of parameter β. As discussed in Section 2.5, as long as the parameter β is sufficiently small, the virtual queue lengths under EGPD are large, roughly of the order $1/\beta$. To see how this affects the holding cost, consider two cases:

(i) If $Q_i(t) \geq 0$, then $\hat{Q}_i(t)$ will also be large (of the order of at least $1/\beta$) since the inequality $\hat{Q}_i(t) \geq Q_i(t)$ holds for all $i \in \mathcal{I}$ at all t.

(ii) If $Q_i(t) < 0$, this has an indirect impact on the holding cost. In particular, large $|Q_i(t)|$ in this case would imply more incomplete matchings. This subsequently results in a higher holding cost.

Therefore, parameter β should be chosen as large as possible, but not to exceed the level beyond which the average rewards start to be significantly (negatively) affected.

Additional queue scaling. Consider arbitrary positive weights γ_i, $i \in \mathcal{I}$. All the results for the EGPD algorithm hold if we use the more general rule

$$j(t) \in \arg\max_{j \in \mathcal{J}} \left[(\partial G(X(t))/\partial X_j)\, w_j + \sum_{i \in \mathcal{I}} \beta\, \gamma_i\, Q_i(t)\, \mu_i(j) \right]. \tag{2.46}$$

instead of (2.3). In this case, it is the weighted vector $(\gamma_i \beta Q_i(t),\, i \in \mathcal{I})$ (not $\beta Q(t)$) that will be close to an optimal dual solution q^*. This property may be used to reduce the holding cost by giving higher weights to more "expensive" queues (with large c_i), thus making them relatively smaller.

Matching completion order. There is a flexibility in choosing which incomplete matching to complete first. For the average matching reward maximization this does not matter (so, earlier we specified the FCFS rule for concreteness). However, if the holding costs are a consideration, one may pick incomplete matchings with higher associated holding cost to be completed first.

2.6.2 Simulation: Average Profit in a Bipartite Matching System

Consider a bipartite matching system, where items arrive in pairs, and the matchings are pairs as well, as is depicted in Figure 2.7. There are 8 item types $\{1, 2, 3, 4, 1', 2', 3', 4'\}$. The arrival graph is on the left, where each edge shows a possible arrival pair, and the plot in the right hand side is the matching graph with edges representing the possible matchings. Up to two matchings can be done per arrival ($m = 2$).

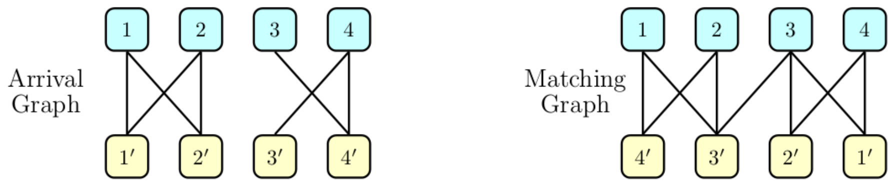


Figure 2.7 Illustration of the matching system.

We consider the process in discrete time $t = 1, 2, \cdots$. The arrival process is i.i.d. across time. Specifically, at each time t, a pair of items enters the system. The probabilities (rates) of different arrival pairs are specified in Table 2.2.

Table 2.2 Probabilities (rates) of different arrival pairs.

Arrival pairs	(1,1′)	(1,2′)	(2,1′)	(2,2′)	(3,4′)	(4,3′)	(4,4′)
Probability	0.166	0.083	0.087	0.083	0.2324	0.2656	0.083

It is easy to check that this system satisfies the necessary and sufficient conditions [15] for bipartite matching systems to be stabilizable. The condition, called NCond in [15], is as follows. Suppose the matching graph is connected. Consider a subset T of items from the top part of the bipartite graph, and denote by α_T the total arrival rate of all items in T. Denote by $B(T)$ the subset of items from the bottom part of the graph that can be matched with at least one item in T, and by $\alpha_{B(T)}$ the total arrival rate of all items in $B(T)$. Then, the system is stabilizable if and only if $\alpha_T < \alpha_{B(T)}$ holds for any *strict* subset T of "top" items.

Now, let us see if Assumption 2.3.5 holds. Since this is a bipartite matching system, with items arriving and departing in pairs, virtual queues satisfy the following linear relation

$Q_1(t) + Q_2(t) + Q_3(t) + Q_4(t) - (Q_{1'}(t) + Q_{2'}(t) + Q_{3'}(t) + Q_{4'}(t)) \equiv 0$. However, given that the NCond condition holds, it is easy to see that Assumption 2.3.5 (formally given by Assumption 2.4.2) holds for this system in the sense described in Section 2.3.6, namely after an orthogonal change of coordinates. (We emphasize again that the algorithm itself remains as is; it does *not* need to do any change of coordinates.) Therefore, the EGPD algorithm is asymptotically optimal for this system for the average reward maximization objective.

Assume linear holding costs, $c \cdot \hat{Q}(t)$, with the cost rate vector $c =$(0.1, 0.2, 0.3, 0.4, 0.4, 0.3, 0.2, 0.1). The matching rewards for different matchings are given in Table 2.3.

Table 2.3 Matching rewards.

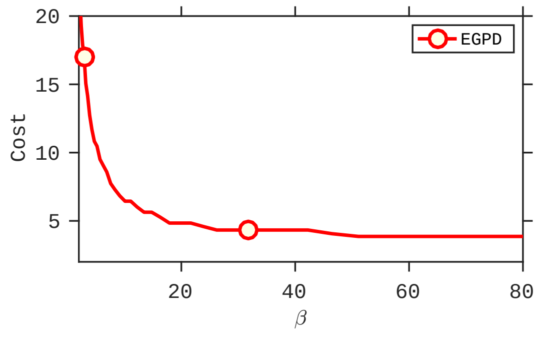
Matchings	$\langle 1,3' \rangle$	$\langle 1,4' \rangle$	$\langle 2,3' \rangle$	$\langle 2,4' \rangle$	$\langle 3,1' \rangle$	$\langle 3,2' \rangle$	$\langle 3,3' \rangle$	$\langle 4,1' \rangle$	$\langle 4,2' \rangle$
Reward	5	50	5	50	5	50	5	50	5

We simulated this system under EGDP scheme. Figure 2.8 shows the dependence of EGPD average performance metrics on the parameter β. The range of β is shown within which the average reward declines from its optimal (largest) value to the “plateau” it reaches when β is large. Parts (a), (b) and (c) show average holding cost, reward and profit, respectively; the average profit is the average reward minus the average holding cost.

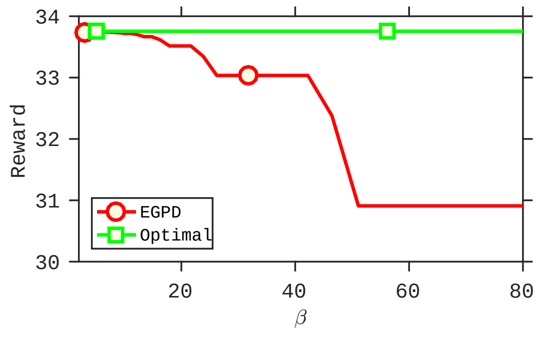
We see that the average profit is maximized within a certain range of values of β, where, roughly speaking, the average reward is “still” close to optimal and the average holding cost is “already” close to the best achievable by EGPD. We conjecture that the average profit with such a choice of β is reasonably close to the optimal profit under any control algorithm. Verifying and quantifying this informal conjecture is an interesting subject for future research.

2.7 Conclusions

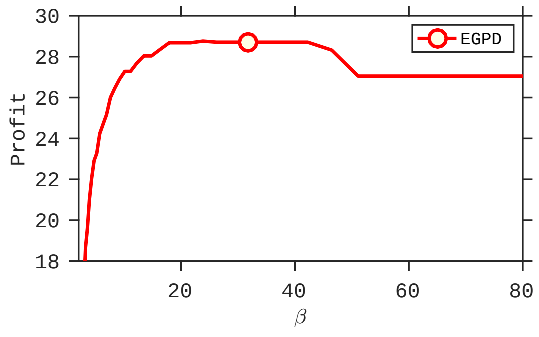
In this chapter, we have proposed an approach for optimal dynamic control of general matching systems. The central idea is using a virtual matching system allowing negative (as well as positive) queues, as part of the overall control scheme. The virtual system fits



(a) Average holding cost



(b) Average reward



(c) Average profit

Figure 2.8 EGPD algorithm performance.

into a queueing network framework, except the queues may be negative, and it is controlled by an extended version of the GPD algorithm, called EGPD. We prove the asymptotic optimality of EGPD. The approach is very generic, not restricted to special cases, such as bipartite matching. The proposed scheme is also very robust in the sense that it does not require the knowledge of input rates, and automatically adapts to changing input rates. Simulations demonstrate good performance of the algorithm.

Although the scheme that we develop has average reward maximization as its objective, the parameter setting can be used to achieve good performance in terms of the more general objective, which includes holding costs. Addressing this and other more general objectives within a dynamic control framework, not requiring a priori knowledge of the item arrival rates, is an important future subject.

Chapter 3

Online Reinforcement Learning: Applications in Customer Journey Optimization

3.1 Introduction

Finding an efficient online control policy is a fundamental problem for many real-world applications [77]. In this work, we propose a framework, that handles this by employing a reinforcement learning (RL) approach and deep neural networks (DNN). The specific type of problem that we target in this chapter is a single-learner agent who concurrently interacts with many instances of an environment with the objective of maximizing the average long-term reward. Having many instances is the key characteristic of this problem that enables us to design an online learning algorithm. The basic idea is simple: learn a good policy by exploration in a subset of the environment instances and then apply the successful experiences to the others.

As depicted in Figure 3.1, we specify a general scheme of the problems which might be handled with the framework proposed in this chapter. A single learner agent is responsible for decision-making. It observes different states of the environment instances in parallel, as well as the rewards obtained in each of them; then, it takes corresponding actions in response to those observations. This problem setting has different applications such as customer journey optimization and predictive maintenance.

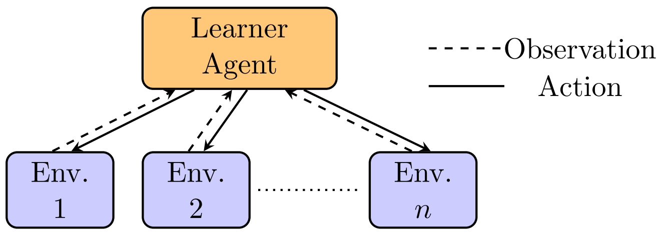


Figure 3.1 Communication of the learner agent with different instances of the environment. These communications might happen in a completely asynchronous manner. The agent learns a policy based on the observations gathered from different environments and is responsible for making decisions in each of them.

Many machine learning algorithms are designed to be trained in a supervised fashion, where a mapping from training inputs to outputs is learned, but supervised learning is not applicable to problems such as customer journey optimization since one does not have information about the "true long-term rewards" as the training labels. One typical choice to achieve this goal is to use reinforcement learning.

Reinforcement learning has gained popularity recently due to the many successful achievements in playing games [43, 61, 79]. It helps the agents to perform tasks in environments that are too complex for humans to determine the correct actions using hand-designed solutions. Most of the recent works have been focused on offline reinforcement learning. The majority of algorithms of this class require having a model that simulates the environment dynamics. For example, in Atari games, an emulator is responsible for refreshing the frames according to the rules of the game and in response to different actions, which yields the next frame and some possible reward. Typically, having an emulator enables the learner agent to play the games many times and learn from various experiences encountered during the training episodes.

However, a major issue of using these class of algorithms in real-world applications is that time progresses sequentially, so this means the agent is dealing with real interactions and it is not possible to propagate a single interaction trajectory through time in order to learn a policy. Unlike games, the model of the environment that the agent interacts with is not known a priori. Although one might think of using an approximate model of the environment to predict the future dynamics, it would still have differences with the real ones. Moreover, most of these models are limited in the sense that they are not able to capture the variability of the environment dynamics. One simple remedy is to periodically update the environment model and re-train the policy with respect to the updated model; however, this method is still problematic since it is unclear how often this process should be triggered and how perfectly the model can reflect the non-stationary elements of the environment.

Another alternative is using model-based RL algorithms [86], which update the model and policy in a unified way. Again, these methods are not convenient in many systems since they usually require a lot of time to first adjust the model based on the dynamics of the new environment and then update the policy. We require an abrupt adaptation to the changes in environment . For example, in online advertisements, some of our displays may not be as well received as we thought, or there can be some material there that could be offensive to some population. We need to learn this information as quickly as we can and update the policy; we should not wait until the customers' attrition to learn. We need to learn within a few interactions without waiting until the end of the time horizon to update the decisions. For this reason, model-free algorithms are the preferred choices for online control since they are learning directly from the environment signals.

Batch-RL algorithms have been common choices for many real-world problems which directly utilize the data to learn a policy, without having the complications related to the design of models; however, one shortcoming of the batch-RL approach is that it does not explore to find new policies. Therefore, the performance of the trained policy is determined by the quality of the data.

An ideal learning agent in the live system should have the capability of learning on-the-fly from various experiences, without requiring any redesign for each new situation it encounters. In fact, it is undesirable to consider the system in which the agent follows a fixed policy over time. For handling these requirements, we introduce the Deep Concurrent Temporal Difference algorithm (DCTD), an extension of Concurrent TD [78] with deep neural networks as our online algorithm. DCTD algorithm remembers the successful control decisions with some instances of the environment so that they can be applied to other ones.

A challenge of deploying an online algorithm is its performance in the initial learning steps, since it needs quite large amount of exploration until identifying a reasonable policy. This shortcoming leaves out the possibility of directly deploying RL algorithms in live systems. To overcome this issue, we add an offline training stage. The goal of this offline stage is to learn a policy so that we can wart start the online RL agent. In this stage, after training an approximate model of the environment, we use an algorithm such as the well-known DQN to train a "reasonable" policy. It is worthwhile to mention that other sophisticated off-policy methods can also be used in the offline stage to find more competitive results, but it is not the focus of this chapter because we only require a policy which does not take irrational decisions.

Figure 3.2 summarizes our proposed implementation framework. This framework suggests a deployment path for our online algorithm that only uses historical data. Unlike many reinforcement learning algorithms that use a model of the task dynamics, DCTD learns online, without a model, and backs up whatever states the agent has encountered during interactions. This direct use of information enables a fast reaction to dynamic changes in the environment and it happens automatically. Lastly, implementing the DCTD algorithm is easy; it does not need any intricate model to predict the result of an action.

In this work, we focus on customer journey optimization as our testbed. We compare the DCTD algorithm with the DQN algorithm. Experimentally, we show a dramatic decline in DQN's performance in exposure to varying environment instances. We observed that the DCTD is robust to these changes, and it automatically adapts without knowing any a priori information about the changes.

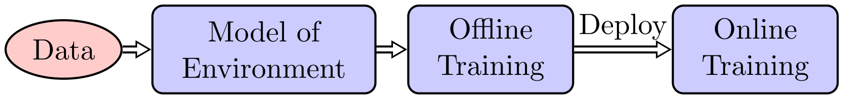


Figure 3.2 The implementation framework: We train a model of the environment using the historical data, which predicts the reward and next state of the environment from a given state and action. The model is employed in an offline algorithm to train a warm-start policy for the deployment. After deployment, the online algorithms will directly interact with the environment instances and uses their real feedback in the training loop.

3.2 Background

In this section, we present the fundamental background that will be extensively used in the rest of this chapter and dissertation.

3.2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a framework for describing sequential decision-making. In the simplest setting, consider an agent that interacts with an environment E. Assuming that the next state of the environment only depends on the current state and action (obviously, independent of the rest of the history), such an interaction can be formalized by a MDP, which can be completely described by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$. Here,

- $\mathcal{S}$ is the *state space* of the environment,
- $\mathcal{A}$ is the *action space* which the agent chooses an action from at every decision point,
- $\mathcal{P}$ is the *state transition probability distribution*, and
- $\mathcal{R}$ is the *reward function* which might be either deterministic or stochastic.

Given the current state $s_t \in \mathcal{S}$ at time t, the agent takes an action a_t from the set of available actions $\mathcal{A}$ with respect to its *policy* π, which is defined as a mapping from $\mathcal{S}$ to $\mathcal{A}$. In response, the agent receives some reward $r_t \sim \mathcal{R}(s_t, a_t)$ and observes the next state s_{t+1} with probability $\mathcal{P}(s_{t+1}|s_t, a_t)$. This cycle continues until achieving a goal of episodic problems or it will run forever in continuous cases. The objective is choosing a sequence of actions that maximizes the cumulative reward.

3.2.2 Reinforcement Learning

For solving MDP problems, one can use any classic *dynamic programming* method, but these methods are usually costly and only applicable to problems with known transition functions. Issues such as the curse of dimensionality limit the use of dynamic programming methods to very small-scale problems. However, MDP provides a theoretical framework upon which reinforcement learning (RL) algorithms are founded. RL is the subfield of machine learning that enables solving these problems. In simple words, they are sample-based algorithms for solving control problems that learn by trial and errors [86]. The agent learns what to do under different conditions based on the previous similar experiences. It learns about good decisions from the signals in the form of a reward that the agent receives from an environment at each decision point.

Let us denote by $R_t = \sum_{j=t}^{\infty} \gamma^{j-t} r_j$ the accumulated discounted reward after time t with discount factor γ. The objective is to find an optimal policy π^*, which maximizes the expected long-term discounted value $Q(s,a) = \mathbb{E}(R_t|s,a)$ from any state s and following policy π^* afterwards. We may consider the policy π as a stochastic policy, which is a conditional distribution $\pi(a|s)$; or it might be considered as a deterministic policy $a = \pi(s)$, which at any given state chooses the decision with probability 1. Our focus in this chapter is only on the deterministic case, but in later chapters we will revisit stochastic policies.

Next, we present an overview of two important RL algorithms on which the offline and online algorithms of this chapter are grounded.

SARSA

SARSA is an on-policy RL algorithm for temporal difference (TD) learning, which iteratively improves the policy being followed by the agent. It uses the update rule (3.1), derived from the Bellman equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right), \tag{3.1}$$

where α is the learning rate. In words, SARSA updates the policy in the direction of reducing the *TD-error*, which is the last term inside the parentheses in the right hand side of (3.1).

Q-learning

Q-learning [99] is a RL algorithm for learning the Q-values for every state-action pair. It iteratively updates the Q-values according to rule (3.2),

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) \right). \tag{3.2}$$

It improves the policy toward the actions with the best performance while the actual actions selected by the policy could be different. These types of algorithms are called off-policy algorithms, i.e., they learn the optimal policy while the agent follows another exploratory policy. One main advantage of this type of update is that the random exploratory actions do not replace the already learned policy, so their learning procedure is usually faster compared to the on-policy ones.

3.2.3 Deep Reinforcement Learning

When the state space is large, it is unavoidable to use function approximators instead of directly calculating $Q(s, a)$ for every state-action pair. In many classical reinforcement learning algorithms, the convergence is achieved when each state-action pair is observed infinitely many times, restricting the applicability of algorithms to only small problems. An alternative is to use a function approximator, which generalizes between similar state-action pairs, so $Q(s, a)$ can be approximately computed for an unseen state s and action a. One can use a function approximator of choice to model the Q-values as $Q(s, a; \theta) \approx Q(s, a)$, where θ is the parameter vector associated with the approximator. Because of their high capability in approximating complex nonlinear functions, *deep neural networks* have recently become been a common choice of a function approximator. In the case of deep neural networks, θ corresponds to a vector of the weights and biases. Once we know the optimal Q^*, the corresponding optimal policy can be extracted simply by

$$\pi^*(s) = \arg\max_a Q^*(s, a; \theta). \tag{3.3}$$

Updates to the network weights θ can be obtained by various reinforcement learning algorithms such as a variant of Q-learning (e.g., the DQN algorithm for Atari games in [61]). DQN utilizes the least knowledge about the environment and learns only by using the sensory information of the environment from the pixel frames and receiving reward signals. In our framework, we have used the same algorithm in the offline pre-deployment stage. In Section 3.4.2, we discuss the details of this algorithm.

3.2.4 Related Works

A framework close to our online algorithm is provided in [62], which uses asynchronous gradient descent to train RL. They consider a central network and multiple agents in which each has its own set of local network parameters. Each of the agents interacts with their own copy of the environment at the same time as the other agents are interacting with theirs. The goal of each agent is to gather independent experiences from the other agents. Combining the agent experiences in the central network leads to a more diverse experience and better learning.

The online algorithm proposed in this chapter is inspired by concurrent reinforcement learning [78]. It is concurrent because it interacts and learns from multiple instances of an environment, and it is online because the policy learning updates occur from an online stream of data. The agent's objective is to learn efficient policies from partial interactions with the environments in flight, trying to maximize the cumulative long-term reward.

3.3 Problem Statement

Consider an agent interacting with a set of environments $\mathcal{E} = \{E^i, i = 1, \cdots, n\}$. We assume that the state space $\mathcal{S}$ and action space $\mathcal{A}$ are equal between all environments, so the dynamics of each environment i can be described by $(\mathcal{S}, \mathcal{A}, \mathcal{P}_t^i, \mathcal{R}_t^i)$. Notice that we allow the transition probabilities and reward function to change over time. We also assume that for every E^i and E^j at time t, the transition probabilities $\mathcal{P}_t^i$ and $\mathcal{P}_t^j$ are independent and equal in distribution, and similarly $\mathcal{R}_t^i$ and $\mathcal{R}_t^j$ are. The independence assumption indicates that the observations (or interactions) in one environment do not affect the behavior of the others.

Typically, it is the case that at every time-step t, the agent is required to take an action only for a subset of environments $\mathcal{E}_t^a \subset \mathcal{E}$, and for the rest of the environments the agent chooses to take no action. For example, in a mailing campaign, the agent might send the offers to the customers located in a certain region; in website visits, the ads can only be presented to the online customers who are currently viewing the web-page; or in predictive maintenance the agent might act only when the attention is on the alerting machines. Similarly, at every time t, the agent updates its policy based on a subset of environments $\mathcal{E}_t^u \subset \mathcal{E}$. Usually, $\mathcal{E}_t^u$ includes the environments for which a reward signal has been observed at time t.

Notice that the set of environments might vary over time, i.e., new environments might be added or removed from $\mathcal{E}$ at different times. To simplify the exposition, here we assume $\mathcal{E}$ is fixed. Notice that considering a variable set $\mathcal{E}$ is a straightforward generalization, and the same framework with minor adjustments applies to this general case.

We denote by τ_t, $t = 0, 1, \cdots$, the sequence of time points when either the agent needs to take an action for an environment or update the policy, i.e., τ_t is the time point where $\mathcal{E}_t^a \cup \mathcal{E}_t^u \neq \emptyset$. From now on, we consider the system as running on discrete time-steps $t = 0, 1, \cdots$. At every time-step t, the agent observes the state of each environment $E^i \in \mathcal{E}_t^a$, denoted by s_t^i, and chooses an action a_t^i from the set of available actions $\mathcal{A}$.

Let us denote by u^i the last time-step before t when an action is chosen for environment E^i. If some reward $r_t^i \sim \mathcal{R}_t^i(s_t, a_t)$ is observed for E^i at time t, by convention, we will assume that it is associated with action a_{u^i}. This assumption is realistic in many real-world applications and it will give all credit of the reward r_t^i to latest action in E^i.

Informally speaking, the objective is to maximize the agent's average long-term reward. This general objective may include various objectives of interest. For example, in customer journey optimization, it can represent various objectives such as a monetary value obtained from the customer, lifetime satisfaction of the customer, or increased frequent visits to the site as long as we can define a sensible reward measure. Customer satisfaction also can be measured through customer reviews and from the number of visits in a specific time window that measures the frequency of visits. In predictive maintenance, the objective can be the maximum availability of the machines or the quality of the products.

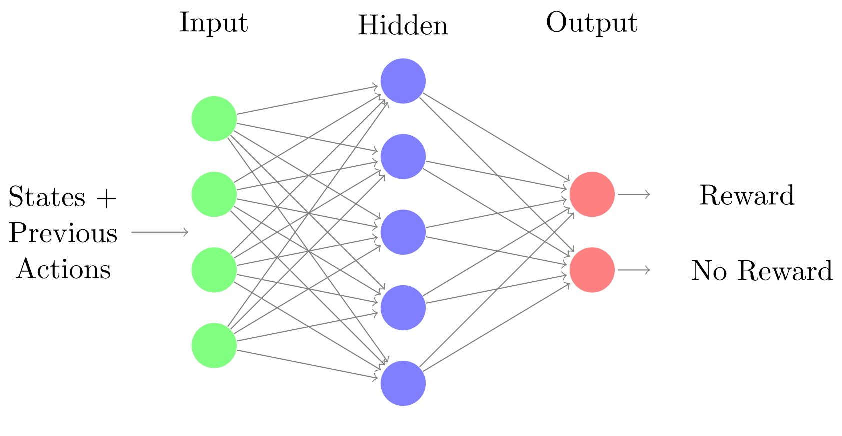
3.4 Proposed Framework

Next, we will describe the details of the proposed framework (Recall Figure 3.2 for an abstract representation of the framework). This deployment path might seem trivial for employing an online RL algorithm, but we are unaware of any previous work using the same approach. One reason might be that there is not too much work done in online RL controls and they are at early stages of development.

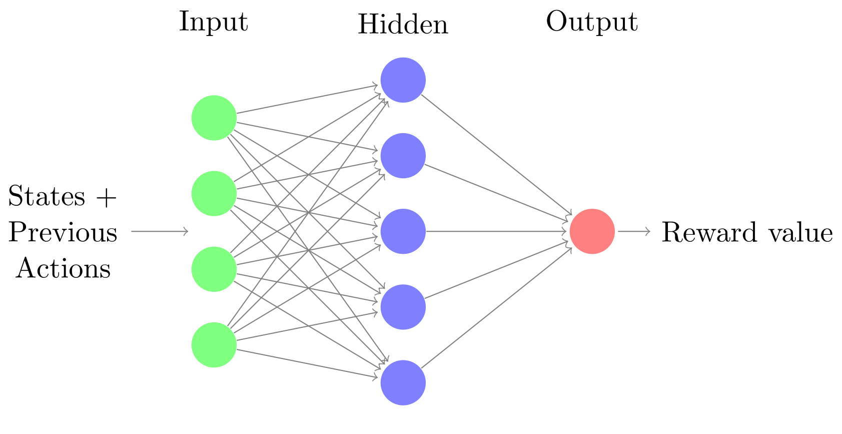
3.4.1 Model of Environment

Many off-policy RL algorithms require a model of the environment, which simulates the environment dynamics. In this section, we specifically introduce a model which uses neural networks for predicting the reward value. Notice that the RL algorithm might or might not directly use the model in the training process. For example model-free algorithms such as SARSA or Q-learning for Atari gamrs only use the visual game screens without knowing how the frames evolve by the internals of the game emulator. But, even in a model-free algorithm, a model is required to be the running engine for producing different experience trajectories.

Figure 3.3 illustrates two neural networks: one classifier and one regressor network. The classifier network specifies whether we predict a positive reward in a certain state. On the other hand, the regressor network helps predicting the reward value for the states with a positive predicted reward. By coupling these two networks, the reward can be predicted for any specific state of the environment. The main advantage of using this structure, instead of having one regressor network, is that it captures the mass probability point at zero (e.g., with a non-zero probability the reward is exactly equal to zero; otherwise its value follows a continuous probability distribution). A similar approach can be utilized for predict the next state s_{t+1} from any given s_t and a_t, but constructing such a prediction model is usually a challenging task, especially for problems with high-dimensional state spaces. An easier practice in many applications such as customer journey optimization is



(a) Classifier network



(b) Regressor network

Figure 3.3 Classifier network (3.3a) specifies whether a reward has occurred or not. In case of any reward, a regressor network (3.3b) determines the value of the reward. By combining the output of these two networks, one may predict the reward that will be obtained in a specific state of the system.

possible; for example, once we have the prediction of the reward r_t, the next state can be uniquely identified, i.e., $s_{t+1} = f(s_t, a_t, r_t)$; we take advantage of this simplification in our experiments.

3.4.2 Offline Algorithm

As discussed before, there is no restriction on type of the algorithms that may be used in this step. Throughout the experiments of this chapter, we have employed a *deep Q-network* (DQN) algorithm [61] in the offline stage. Next, we will briefly describe the details of this algorithm.

Using an extension of Q-learning with neural network function approximators for the Q function, the DQN algorithm in [61] is shown to achieve super-human performance on learning complex policies for Atari games. It iteratively tries to minimize a loss function

$$\mathcal{L}_i(\theta_i) = \mathbb{E}\left[r + \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right]^2, \tag{3.4}$$

where s' is the next state after observing state s.

It was long believed that neural networks make RL algorithms unstable, but DQN used two techniques to ensure the stability of the learning process. First, they store a lot of experience (s_t, a_t, r_t, s_{t+1}) in a buffer called *replay memory.* During training, the experiences are drawn from this memory uniformly at random and used for updating the Q-values. This selection procedure decorrelates the current state of the agent from the experiences used for updating Q-values. This way, the learning environment is almost stationary, which is a necessary condition for convergence of Q-leaning. The second modification uses a *target network* for producing the target values in addition to the main policy network. The target network has the same structure as the main network; the only difference is that the target network is updated with a lower frequency (e.g., every K training iteration, the weights of the target network are synchronized with those of the main network).

3.4.3 Online Algorithm

Our online algorithm is an extension of the concurrent TD algorithm by [78]. Their paper presents the basic ideas and has left plenty of possible extensions untouched, some of which we address here. First, they explore every single environment one by one and learn from them individually. This approach might be very noisy when the environments have skewed reward distributions or mass points with outlier rewards. For example, the network update by the gradient of one outlier observation may degrade a favorable policy, which was performing well in the majority of environments. This update method can also be computationally expensive for a highly concurrent environment with large $\mathcal{E}_t^u$, since it requires a separate training pass for each environment. The computational efficiency will become even worse by employing more sophisticated function approximators for estimating the Q-values. Lastly, they used linear approximators, which do not have enough capacity to represent complex policies.

We address these issues in our algorithm. At each time t, we take multiple batches with replacement from $\mathcal{E}_t^u$ for updating the policy. This batching method causes some of the environments to appear in multiple batches. Instead, one might think of choosing a disjoint batch of environments at every time, i.e., every environment can be selected in only one $\mathcal{B}_t^j$ at every t. We adopted the first approach since it would be more data efficient. Our experiments show that with replacements, we gained almost the same amount of information with less exploration which yielded a higher reward. If an environment is chosen in multiple batches, we will only apply the last suggested action at time t, since it comes from the most recently updated policy.

The agent uses ϵ-*greedy* rule to pick the actions, meaning that the actions are chosen from a random policy with probability ϵ and otherwise according to the agent's policy. Unlike the offline RL algorithms, ϵ is fixed over time. Our suggestion is to choose ϵ not too big to avoid a lot of random actions and not too small to limit the exploratory behavior of the algorithm.

We used neural networks as a nonlinear function approximator of $Q(s, a)$. Similar to the DQN algorithm [61], the combination of vanilla concurrent TD algorithm [78] with deep neural networks is fundamentally unstable. We resolve this problem by employing a non-trainable target network with weights θ^- whose weights are synchronized with the original network every K training steps.

DCTD uses multi-step discounted propagation of the rewards between successive time-steps when an environment is chosen for updating the policy. At every time t and using batch $\mathcal{B}_t^u$, it updates the policy weights in the direction of reducing the TD error, i.e.,

$$\begin{aligned}\Delta\theta = \frac{\alpha}{|\mathcal{B}_t^u|} \sum_{\{i:E_i \in \mathcal{B}_t^u\}} \big[R^i_{u^i:t} + \gamma^{\tau_t - \tau_{u^i}} Q(s_t^i, a_t^i; \theta^-) \\ - Q(s^i_{u^i}, a^i_{u^i}; \theta)\big] \nabla_\theta Q(s^i_{u^i}, a^i_{u^i}; \theta), \end{aligned} \tag{3.5}$$

where $R^i_{u^i:t} = \sum_{k=u^i}^{t} \gamma^{\tau_k - \tau_{u^i}} r_k^i$.

Algorithm 3 DCTD Algorithm

```
 1: initialize a network $Q$ with random weights $\theta$.
 2: initialize a target network $Q$ with weights $\theta^- \leftarrow \theta$
 3: $R^i \leftarrow 0$, $u^i \leftarrow 0$ for every $E^i \in \mathcal{E}$
 4: for every time-step $t$ do
 5:     $a_t^i \leftarrow \varnothing$ for every $E^i \in \mathcal{E}$
 6:     for training steps $j = 1, \cdots, b_t$ do
 7:         take batch of environments $\mathcal{B}_t^j \subset \mathcal{E}_t^u$
 8:         if  environment $E^i \in \mathcal{B}_t^j \cap \mathcal{E}_t^a$ then
 9:             choose action $a_t^i \leftarrow \epsilon$-greedy(Q)
10:         end if
11:         for environment $E^i \in \mathcal{B}_t^j$ do
12:             set $y^i = R^i + \gamma^{t-u^i} Q(s_t^i, a_t^i; \theta^-)$
13:         end for
14:         perform a gradient descent step on
                    $\big(y^i - Q(s^i_{u^i}, a^i_{u^i}; \theta)\big)^2$ with respect to $\theta$
15:     end for
16:     $a_t^i \leftarrow$greedy(Q) for every $E^i \in \mathcal{E}_t^a \setminus \bigcup_{j=1}^{b_t} \mathcal{B}_t^j$
17:     Execute actions $a_t^i$, get reward $r_t^i$ and observe $s_{t+1}^i$ for all $i$
18:     for every $E^i \in \mathcal{E}$ with $r_t^i > 0$ do
19:         $R^i \leftarrow R^i + \gamma^{\tau_t - \tau_{u^i}} r_t^i$,
20:     end for
21:     $R^i \leftarrow 0$, $u^i \leftarrow t$ for every $E^i \in \mathcal{E}_t^u$
22:     Every $K$ training step, $\theta^- \leftarrow \theta$.
23: end for
```

Comparison to Mnih et al. [62]: Our algorithm is closely related to the approach used in [62], but in a different paradigm. We show that by taking advantage of having a large number of environments in parallel, we can train a policy online. Provided with enough information coming through a large number of diverse environments, we can have multiple training steps at every time-step, allowing an online reaction to the recent environment dynamics.

3.5 Experiment: Customer Journey Optimization

In this section, we show how our proposed framework can be utilized in a real-world application. Specifically, we focus on an "outbound" customer journey optimization in our case study, where a firm initiates different communications to its customers through marketing channels. An example of this case is a firm that reaches out to its customers by sending them advertisements via different portals such as emails, mails, or phone calls. A fundamental challenge of the firm is that it not only wants customers to buy the products, but also it wants to develop a healthy relationship with the customer base. To achieve this goal, the firm requires a strong and efficient policy in maintaining and creating relationships with customers. This problem becomes even more complicated with diverse customer bases, which might have very different reactions to the different communications, so the firm should be very careful at each interaction point. By following a wise marketing strategy, the firm has the opportunity to influence the customers' future preferences, effectively change their reactions in the next interaction points, motivate them to campaign, and keep buying more products and services.

To meet these requirements, we formulate the customer journey optimization with an RL approach which tries to maximize the lifetime value of customers, instead of only considering myopic immediate rewards. The proposed framework provides an interactive dynamic learning algorithm while keeping track of the customer reactions and adapting accordingly to their behavioral changes.

Using Reinforcement Learning for the customer journey problem is still in its early stages. Most of the previous literature has been pursuing an off-policy RL approach [1, 70, 89, 90, 92]. The first attempt is by [70] in which the authors used a batch Q-learning algorithm for solving a mailing campaign problem. Tkachenko [92] applies an algorithm similar to DQN for approaching a good policy. Silver et al. [78] argue that learning offline from the complete customers trajectories is not efficient. They introduced the concurrent TD algorithm which learns a good policy from customers' partial interactions. One of the challenges in the customer journey optimization problem is the partial observability of the customers since the agent only has access to noisy and incomplete information about the state. In order to address this problem, [53] proposed a hybrid approach which models the environment by using recurrent neural networks and LSTM in a supervised learning manner. They couple this network with another neural network that learns the optimal policy by applying a DQN algorithm. Even though using recurrent neural networks captures more historical information about the customers, their approach is not a straightforward end-to-end training. Capturing long-term dependencies is not the focus of this work, but rather we will keep track of the customers' history simply through using multi-period inputs.

The mapping of the customer journey optimization framework to the proposed model is straightforward. The agent is the firm that concurrently deals with multiple customers, where each customer is a separate instance of the environment. Next, we specifically describe the experiment details.

3.5.1 Experiment Setup

In this experiment, we use a real marketing dataset, KDD1998, which is a common sample in the marketing literature [69]. This specific data is for an outbound mailing campaign in a non-profit organization. The data includes the interaction history of the organization for twenty-three periods. The action set includes 11 mailing types and 1 action that represents doing nothing. The reward is defined as the donations that the organization receives.

Each customer record has variables that describe the interaction history. The state of the customers can be identified by two sets of variables. The variables of the first category, which we call "dynamic variables," change according to the organization's actions and the customer reactions as they progress through time. Specifically, we consider the following well-known variables in the literature [93]: recency, frequency, average monetary value, interaction recency, and interaction frequency. The second category includes "static variables" such as demographic information, which captures different segments of the customer base. We use age, zip region, gender, and income group as the static variables. In addition, we use the previous actions and previous rewards in defining the internal state of the learning algorithm.

Using this data, we train a model similar to [93]. In a given state of the customer, this model simulates the customer's behavior in response to an action. The customer's model is trained in a supervised fashion before starting the reinforcement learning training. During RL training the model is simply scored, and it is not trainable. We use one deep neural network for predicting the donation amount in a given state of the customer and another one for classifying the occurrence of a donation, as illustrated in Section 3.4.1. By combining the output of these two networks using the method described in [93], the reward amount r_t will be identified. To find the next state s_{t+1}, we only update the dynamic variables according to r_t and s_t. Since we don't have detailed customer information such as date of birth, we let the static variables be fixed over time. We encourage interested readers to check the details of model construction and state transition formulas in [93]. Since we trained the model directly from data, it generates a reasonable approximation for the real-life interaction dynamics.

The offline stage helps us learn a policy that takes into account the entire customer journey. In this stage, we take advantage of the model and interact with each one of the customers until the end of the marketing campaign (i.e., 23 periods) before moving on to the next customer. This way, by using the DQN algorithm, we learn from the customer's entire trajectory. It is worthwhile to mention that we have also employed techniques such as using an experience replay and target network that DQN uses to ensure the stability of the learning process.

Once we train a reasonable policy, we use it as a starting policy of the online algorithm. We want to point out that the model might not represent the "real" customers' behaviors, and as a result, the policy trained using the model might differ from the optimal one; however, we neglect these inaccuracies in the hope that the online algorithm would capture them. Therefore, we initialize the DCTD's network with the trained DQN network and train it in the online setting, too.

Even though it is not possible to deploy the case study in the real world, we evaluate the effectiveness of DCTD in live systems by designing a few tests. Specifically, we simulate the "live" system by using a modified model of the customers, where the donation probability of some of the actions was increased, so the live system would return more donation amount to those actions on average. Hereinafter, we will refer to the percentage of the increase in donation probabilities by *noise*. The DCTD algorithm should be able to learn these noises and exploit them to maximize the lifetime value of the customers.

In order to see the value of initializing the DCTDs network with the trained DQN network, we also provide the results of the DCTD algorithm with the randomly initialized weights. This comparison provides insights about the role of the pre-trained DQN network.

3.5.2 Results

This section presents the experimental results. Our DNN network in both the DQN and DCTD algorithms is a fully connected network with four hidden layers and ReLU activations. In order to capture the interaction history, we use the state of the customer, the previous actions, and the previously observed rewards of the last three periods as the input for our DNN. The output is the Q-value for each of the 12 possible actions. The DQN network is trained with 100,000 customers, each of them propagated for 23 periods. The DCTD algorithm was also concurrently trained with 153,000 customers. We also use a mini-batch of 256 customers at every time-step of the DCTD algorithm, and $b_t = 600$. In DCTD, we also set $\gamma = 0.99$ and $\epsilon = 0.1$.

In order to see the effectiveness of our algorithms, we compare our results with that of three other benchmark policies: (*i*) the original policy, which uses the actions stored in the original dataset, (*ii*) a random policy, in which the action for each customer in each period is picked randomly; and (*iii*) a myopic policy that chooses actions with the maximum immediate reward. In order to call the myopic policy, we use the model of the environment to evaluate all actions and choose the one with the highest reward. Note that this policy does not take long-term rewards into account.

Results of DQN: As illustrated in Figure 3.4, the policy learned by the DQN algorithm obtains a higher long-term reward compared to the other policies. It outperforms the random, original actions, and myopic policies by 65%, 69%, and 20%, respectively. A comparison of DQN with the myopic policy reflects the performance boost that we can gain by using an RL approach over using supervised learning, which only considers the immediate reward. This fact implies that the myopic approach is not the best formulation for customer journey optimization. We also observed that the cumulative donation of the original actions was not too different from (or, even worse than) the ones obtained by using the random policy. This observation raises a suspicion about the existence of any non-random policy in the organization's original marketing.

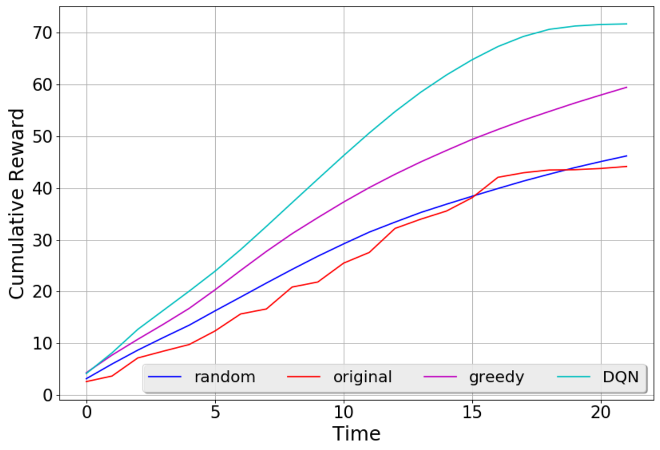


Figure 3.4 Comparison of the donation obtained from DQN algorithm with the myopic, random, and original action policies.

Results of DCTD: For simulating the live system in our experiments, as discussed before, we use a modified model where the donation probabilities of actions 2, 4, and 6 have been increased by 20%. These actions were chosen randomly from the set of actions. We ran the live system for another marketing campaign consisting of 22 periods. In the following experiments, the live system was incorporated right from time 0.

Figure 3.5 shows the results of different algorithms running in the live system. We observe that DCTD with the DQN policy initialization follows the DQN path at the beginning, but it quickly detects live system dynamics. By adapting its policy, DCTD is able to provide far better cumulative rewards compared to the other offline algorithms, i.e., greedy, random policy and DQN. Since we do not update the DQN policy in the live system, this comparison also shows how well the DCTD learns the customer behaviors that were not captured by the model. Another interesting observation in this figure is the performance of DQN, which is worse than the myopic policy. This observation follows from the fact that the noisy actions were less likely to be selected by DQN and provides evidence that a policy trained with an offline RL algorithm might fail in live systems.

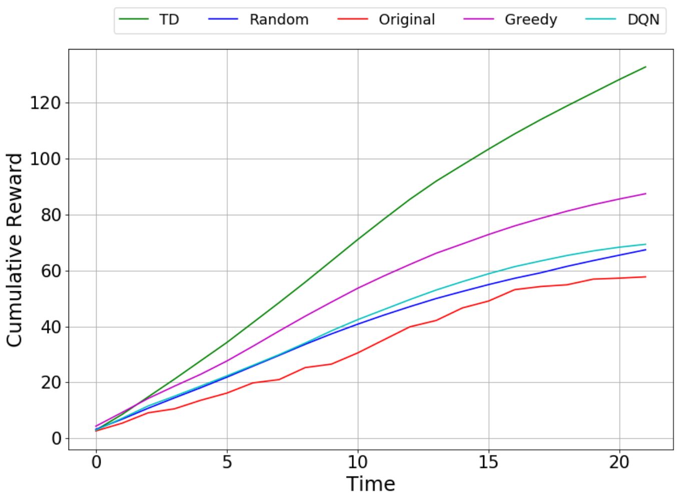


Figure 3.5 A comparison of different algorithms in the simulated live environment, which was constructed by adding 20% to the donation probabilities of action 2, 4, and 6 in the model.

In order to see the impact of different noise levels, we test live environments with $\{5, 10, \ldots, 95\}$ percent noises. Figure 3.6 presents the corresponding results, in which, in addition to the previous algorithms, the results of the DCTD algorithm with random initialization is also added, with label DCTD. Also, the result of the DCTD algorithm initialized with the DQN's policy is provided, with label DCTD-DQN.

Figure 3.6a shows the result with noisy actions {2, 4, 6}, and Figure 3.6b presents the result for noisy actions {3, 5, 10}. As seen in both, DCTD and DCTD-DQN can capture even a small noise level of 5%, which is usually a hard task. With this increase in the noise level, we observe that the cumulative reward values get larger, until they become almost fixed after a threshold around 50%. This behavior is due to the method by which the model combines the donation probability with its corresponding donation value. When the donation probability is larger than, say, 50%, the role of the classifier network is negligible. As Figure 3.6a shows, DCTD-DQN provides slightly better results than DCTD for small noises, but for large ones, they had almost the same performance. This gap is larger in

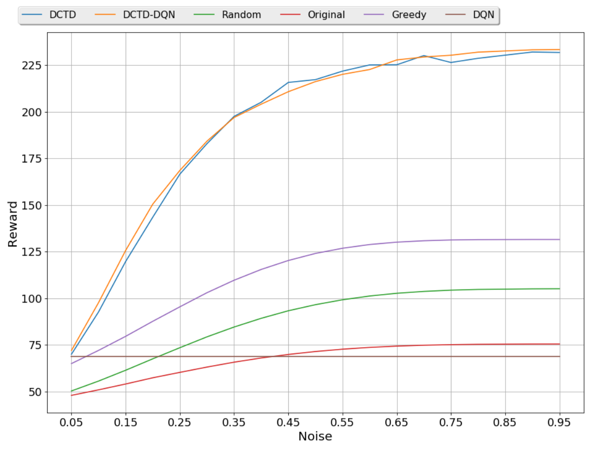
Figure 3.6b and DCTD-DQN obtains around 10% more reward compared to DCTD, which testifies to the strength and importance of pre-training. In addition, it is worth mentioning that without the pre-training phase, one does not have any idea about the structure of the neural network, and it needs some parameter tuning too, which is not possible in the live system.

Finally, Figure 3.7 presents the total number of times when one of the noisy actions 2, 4, or 6 is selected by the DCTD and DQN algorithms over time. As we see, the DCTD learns the noise after few iterations and exploits this knowledge up to the end of the campaign. However, since the DQN policy is fixed in the live system, it was unaware of those desirable actions and chose a number of them.

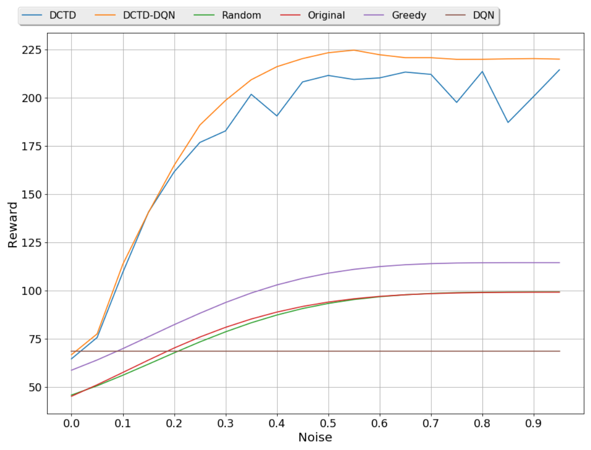
3.6 Other Applications: Predictive Maintenance

Motivated by recent advances in the Internet of Things, we will discuss on how the proposed framework can be applied to optimize automated maintenance decisions in a network with connected devices. Consider a system that is composed of many similar devices that are connected to a central unit through a network. The devices can operate in different physical locations or in close proximity. Each device can talk to a central unit and share information back and forth. The central unit concurrently gathers the sensors' data from each device and is the main location where a maintenance plan can be updated and/or designed. The central unit tries to choose maintenance actions in order to optimize the goal of the system (e.g., minimizing the long-term maintenance cost or maximizing the network uptime).

A common practice to obtain a maintenance plan of such a network is following periodic inspections and emergency maintenance. These approaches can be very expensive, especially when the functionality of each device is crucial in a way that its failure might cause a whole network to shut down. With the emergence of big data, the corresponding analytical tools, and computational power attracted more focus on predictive and proactive maintenance, which uses the sensory information of devices to design wise maintenance plans. Arguably, our proposed framework can be applied to this problem as well. This policy is deployed online in the central unit, and each connected device contributes its own share of data to



(a) different noise on actions 2, 4, and 6



(b) different noise on actions 3, 5, and 10

Figure 3.6 Reaction of the online algorithm to different levels of the noise in donation probability. We observed that DCTD reacted for even very small noise at 5%, which is hard to capture.

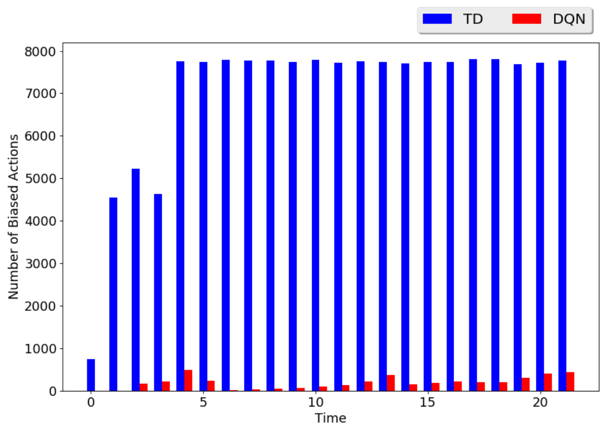


Figure 3.7 How the DCTD action selection procedure adapts to the live system over time. Each bar represents the cumulative number of times when these noisy actions are selected by a different algorithm. It takes approximately 4 time-steps for the DCTD algorithm to fully explore the new policy.

the central decision-making unit so that new updates can be made to the policy to reflect real-life operating conditions. By learning from the failures of different devices, the central unit chooses predictive maintenance actions to prevent high-risk situations. This way, the online policy learns concurrently from different devices working in presumably different operating conditions and decisions are made automatically.

3.7 Discussion and Conclusion

In this work, we addressed some of the issues that hinder the use of RL in practical deployments for real-world problems where accuracy and adaptiveness are essential. We focused on a set of problems in which an agent interacts, in parallel, with many instances of an environment and presented a framework to deploy an online learner agent. We introduced an extension of concurrent TD, namely DCTD, which learns online from partial interactions with the environment instances. The strength of the DCTD algorithm is that it learns good policies, in realtime, without knowing the environment dynamics, and only by observing the experiences of different environments.

Our case study considers an outbound marketing campaign, but as we stated before, this approach is not confined to this specific problem. The same framework and algorithms may be applied, with minor adjustments, to other applications of interest, such as inbound marketing (i.e., the customers are the initiators of the communication) or predictive maintenance.

Chapter 4

Reinforcement Learning for Solving the Vehicle Routing Problem

4.1 Introduction

The *Vehicle Routing Problem* (VRP) is a combinatorial optimization problem that has been studied in operations research and computer science for decades. VRP is known to be a computationally difficult problems for which many exact and heuristic algorithms have been proposed, but providing fast and reliable solutions is still a challenging task. In the simplest form of the VRP, a single capacitated vehicle is responsible for delivering items to multiple customer nodes; the vehicle must return to the depot to pick up additional items when it runs out. The objective is to optimize a set of routes, all beginning and ending at a given node, called the *depot*, in order to attain the maximum possible reward, which is often the negative of the total vehicle distance or average service time. This problem is computationally difficult to solve to optimality, even with only a few hundred customer nodes [29], and is classified as an NP-hard problem [52]. For an overview of the VRP, see, for example, [35, 49, 50, 94].

The prospect of new algorithm discovery, without any hand-engineered reasoning, makes neural networks and reinforcement learning a compelling choice that has the potential to be an important milestone on the path toward solving these problems. In this chapter, we develop a framework with the capability of solving a wide variety of combinatorial optimization problems using *Reinforcement Learning* (RL) and show how it can be applied to solve the VRP. For this purpose, we consider the Markov Decision Process (MDP) formulation of the problem, in which the optimal solution can be viewed as a sequence of decisions. This allows us to use RL to produce near-optimal solutions by increasing the probability of decoding "desirable" sequences.

A naive approach would be to train an instance-specific policy by considering every instance separately. In this approach, an RL algorithm needs to take many samples, maybe millions of them, from the underlying MDP of the problem to be able to produce a good-quality solution. Obviously, this approach is not practical since the RL method should be comparable to existing algorithms not only in terms of solution quality but also in terms of runtime. For example, for all of the problems studied in this chapter, we wish to have a method that can produce near-optimal solutions in less than a second. Moreover, the policy learned by this naive approach would not apply to instances other than the one that was used in the training; after a small perturbation of the problem setting, e.g., changing the location or demand of a customer, we would need to rebuild the policy from scratch.

Therefore, rather than focusing on training a separate policy for every problem instance, we propose a structure that performs well on any problem sampled from a given distribution. This means that if we generate a new VRP instance with the same number of nodes and vehicle capacity, and the same location and demand distributions as the ones that we used during training, then the trained policy will work well, and we can solve the problem right away, without retraining for every new instance. As long as we approximate the generating distribution of the problem, the framework can be applied. One can view the trained policy as a black-box heuristic (or a meta-algorithm) which generates a high-quality solution in a reasonable amount of time.

This study is motivated by the recent work by Bello et al. [8]. We have generalized their framework to include a wider range of combinatorial optimization problems such as the VRP. Bello et al. [8] propose the use of a Pointer Network [96] to decode the solution. One major issue that complicates the direct use of their approach for the VRP is that it assumes the system is static over time. In contrast, in the VRP, the demands change over time in the sense that once a node has been visited its demand becomes, effectively, zero. To overcome this, we propose an alternate approach—which is simpler than the Pointer Network approach—that can efficiently handle both the static and dynamic elements of the system. Our policy model consists of a recurrent neural network (RNN) decoder coupled with an attention mechanism. At each time step, the embeddings of the static elements are the input to the RNN decoder, and the output of the RNN and the dynamic element embeddings are fed into an attention mechanism, which forms a distribution over the feasible destinations that can be chosen at the next decision point.

The proposed framework is appealing to practitioners since we utilize a self-driven learning procedure that only requires the reward calculation based on the generated outputs; as long as we can observe the reward and verify the feasibility of a generated sequence, we can learn the desired meta-algorithm. For instance, if one does not know how to solve the VRP but can compute the cost of a given solution, then one can provide the signal required for solving the problem using our method. Unlike most classical heuristic methods, it is robust to problem changes, e.g., when a customer changes its demand value or relocates to a different position, it can automatically adapt the solution. Using classical heuristics for VRP, the entire distance matrix must be recalculated and the system must be re-optimized from scratch, which is often impractical, especially if the problem size is large. In contrast, our proposed framework does not require an explicit distance matrix, and only one feed-forward pass of the network will update the routes based on the new data.

Our numerical experiments indicate that our framework performs significantly better than well-known classical heuristics designed for the VRP, and that it is robust in the sense that its worst results are still relatively close to optimal. Comparing our method with the OR-Tools VRP engine [36], which is one of the best open-source VRP solvers, we observe a noticeable improvement; in VRP instances with 50 and 100 customers, our method provides shorter solutions in roughly 61% of the instances. Another interesting observation that we make in this study is that by allowing multiple vehicles to supply the demand of a single node, our RL-based framework finds policies that outperform solutions that require single deliveries. We obtain this appealing property, known as *split delivery*, without any hand engineering and at no extra computational cost. We show that with minor changes to the structure, a similar framework can applied to solve more complicated *stochastic VRP* (SVRP) problems.

4.2 Background

Before presenting the problem formalization, we briefly review the required notation and relation to existing work.

Sequence-to-Sequence Models *Sequence-to-Sequence* models [57, 85, 96] are useful in tasks for which a mapping from one sequence to another is required. They have been extensively studied in the field of neural machine translation over the past several years, and there are numerous variants of these models. The general architecture, which is shared by many of these models, consists of two RNN networks called the encoder and decoder. An encoder network reads through the input sequence and stores the knowledge in a fixed-size vector representation (or a sequence of vectors); then, a decoder converts the encoded information back to an output sequence.

In the vanilla Sequence-to-Sequence architecture [85], the source sequence appears only once in the encoder and the entire output sequence is generated based on one vector (i.e., the last hidden state of the encoder RNN). Other extensions, for example Bahdanau et al. [7], illustrate that the source information can be used more explicitly to increase the amount of information during the decoding steps. In addition to the encoder and decoder networks, they employ another neural network, namely an *attention mechanism* that *attends* to the entire encoder RNN states. This mechanism allows the decoder to focus on the important locations of the source sequence and use the relevant information during decoding steps for producing "better" output sequences. Recently, the concept of attention has been a popular research idea due to its capability to align different objects, e.g., in computer vision [18, 40, 102, 103] and neural machine translation [7, 42, 57]. In this study, we also employ a special attention structure for the policy parameterization. See Section 4.3.3 for a detailed discussion of the attention mechanism.

Neural Combinatorial Optimization Over the last several years, multiple methods have been developed to tackle combinatorial optimization problems by using recent advances in artificial intelligence. The first attempt was proposed by Vinyals et al. [96], who introduce the concept of a *Pointer Network*, a model originally inspired by sequence-to-sequence models. Because it is invariant to the length of the encoder sequence, the Pointer Network enables the model to apply to combinatorial optimization problems, where the output sequence length is determined by the source sequence. They use the Pointer Network architecture in a supervised fashion to find near-optimal Traveling Salesman Problem (TSP) tours from ground truth optimal (or heuristic) solutions. This dependence on supervision prohibits the Pointer Network from finding better solutions than the ones provided during the training.

Closest to our approach, Bello et al. [8] address this issue by developing a neural combinatorial optimization framework that uses RL to optimize a policy modeled by a Pointer Network. Using several classical combinatorial optimization problems such as TSP and the knapsack problem, they show the effectiveness and generality of their architecture.

On a related topic, Dai et al. [27] solve optimization problems over graphs using a graph embedding structure [26] and a deep Q-learning (DQN) algorithm [61]. Even though VRP can be represented by a graph with weighted nodes and edges, their proposed approach does not directly apply since in VRP, a particular node (e.g., the depot) might be visited multiple times.

Next, we introduce our model, which is a simplified version of the Pointer Network.

4.3 The Model

In this section, we formally define the problem and our proposed framework for a generic combinatorial optimization problem with a given set of inputs $X \doteq \{x^i, i = 1, \cdots, M\}$. We allow some of the elements of each input to change between the decoding steps, which is, in fact, the case in many problems such as the VRP. The dynamic elements might be an artifact of the decoding procedure itself, or they can be imposed by the environment. For example, in the VRP, the remaining customer demands change over time as the vehicle visits the customer nodes; or we might consider a variant in which new customers arrive or adjust their demand values over time, independent of the vehicle decisions. Formally, we represent each input x^i by a sequence of tuples $\{x^i_t \doteq (s^i, d^i_t), t = 0, 1, \cdots\}$, where s^i and d^i_t are the static and dynamic elements of the input, respectively, and can themselves be tuples. One can view x^i_t as a vector of features that describes the state of input i at time t. For instance, in the VRP, x^i_t gives a snapshot of the customer i, where s^i corresponds to the 2-dimensional coordinate of customer i's location and d^i_t is its demand at time t. We will denote the set of all input states at a fixed time t with X_t.

We start from an arbitrary input in X_0, where we use the pointer y_0 to refer to that input. At every decoding time t ($t = 0, 1, \cdots$), y_{t+1} points to one of the available inputs X_t, which determines the input of the next decoder step; this process continues until a termination condition is satisfied. The termination condition is problem-specific, showing that the generated sequence satisfies the feasibility constraints. For instance, in the VRP that we consider in this work, the terminating condition is that there is no more demand to satisfy. This process will generate a sequence of length $T+1$, $Y = \{y_t, t = 0, ..., T\}$, possibly with a different sequence length compared to the input length M. This is due to the fact that, for example, the vehicle may have to go back to the depot several times to refill. We also use the notation Y_t to denote the decoded sequence up to time t, i.e., $Y_t = \{y_0, \cdots, y_t\}$. We are interested in finding a stochastic policy π which generates the sequence Y in a way that minimizes a loss objective while satisfying the problem constraints. The optimal policy π^* will generate the optimal solution with probability 1. Our goal is to make π as close to π^* as possible. Similar to Sutskever et al. [85], we use the probability chain rule to decompose the probability of generating sequence Y, i.e., $P(Y|X_0)$, as follows:

$$P(Y|X_0) = \prod_{t=0}^{T} \pi(y_{t+1}|Y_t, X_t), \tag{4.1}$$

and

$$X_{t+1} = f(y_{t+1}, X_t) \tag{4.2}$$

is a recursive update of the problem representation with the state transition function f. Each component in the right-hand side of (4.1) is computed by the attention mechanism, i.e.,

$$\pi(\cdot|Y_t, X_t) = \text{softmax}(g(h_t, X_t)), \tag{4.3}$$

where g is an affine function that outputs an input-sized vector, and h_t is the state of the RNN decoder that summarizes the information of previously decoded steps $y_0, \cdots, y_t$. We will describe the details of our proposed attention mechanism in Section 4.3.3.

Remark 1: This structure can handle combinatorial optimization problems in both a more classical static setting as well as in dynamically changing ones. In static combinatorial optimization, X_0 fully defines the problem that we are trying to solve. For example, in the VRP, X_0 includes all customer locations as well as their demands, and the depot location; then, the remaining demands are updated with respect to the vehicle destination and its load. With this consideration, often there exists a well-defined Markovian transition function f, as defined in (4.2), which is sufficient to update the dynamics between decision points. However, our framework can also be applied to problems in which the state transition function is unknown and/or is subject to external noise, since the training does not explicitly make use of the transition function. However, knowing this transition function helps in simulating the environment that the training algorithm interacts with. See Section 4.4.7 for an example of how to handle a stochastic version of the VRP in which random customers with random demands appear over time.

4.3.1 Limitations of Pointer Networks

Although the framework proposed by Bello et al. [8] works well on problems such as the knapsack problem and TSP, it is not efficient for more complicated combinatorial optimization problems in which the system representation varies over time, such as VRP. Bello et al. [8] feed a random sequence of inputs to the RNN encoder. Figure 4.1 illustrates with an example why using the RNN in the encoder is restrictive. Suppose that at the first decision step, the policy sends the vehicle to customer 1, and as a result, its demand is satisfied, i.e., $d_0^1 \neq d_1^1$. Then in the second decision step, we need to re-calculate the whole network with the new d_1^1 information in order to choose the next customer. The dynamic elements complicate the forward pass of the network since there should be encoder/decoder updates when an input changes. The situation is even worse during back-propagation to accumulate the gradients since we need to remember when the dynamic elements changed. In order to resolve this complication, we require the policy model to be *invariant to the input sequence* so that changing the order of any two inputs does not affect the network. In Section 4.3.2, we present a simple network that satisfies this property.

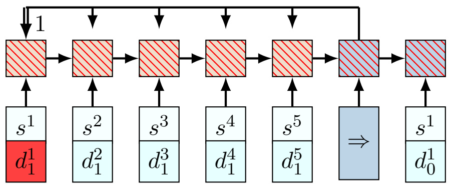


Figure 4.1 Limitation of the Pointer Network. After a change in dynamic elements (d_1^1 in this example), the whole Pointer Network must be updated to compute the probabilities in the next decision point.

4.3.2 The Proposed Neural Network Model

We argue that the RNN encoder adds extra complication to the encoder but is actually not necessary, and the approach can be made much more general by omitting it. RNNs are necessary only when the inputs convey sequential information; e.g., in text translation the combination of words and their relative position must be captured in order for the translation to be accurate. But the question here is, *why do we need to have them in the encoder for combinatorial optimization problems when there is no meaningful order in the input set?* As an example, in the VRP, the inputs are the set of unordered customer locations with their respective demands, and their order is not meaningful; any random permutation contains the same information as the original inputs. Therefore, in our model, we simply leave out the encoder RNN and directly use the embedded inputs instead of the RNN hidden states. By this modification, many of the computational complications disappear, without decreasing the effectiveness. In Section 4.4.1, we provide experiments to verify this claim.

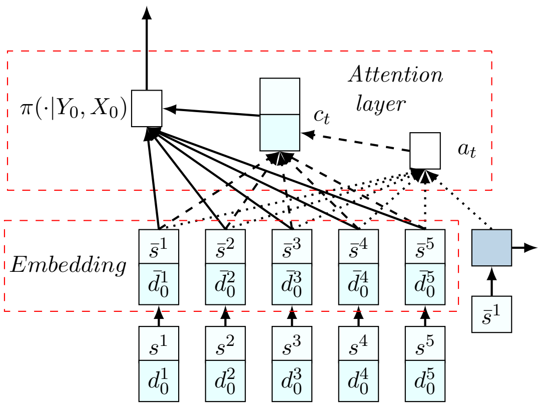

$\pi(\cdot|Y_0, X_0)$
c_t
a_t
Attention layer
Embedding
$\bar{s}^1$ $\bar{d}_0^1$ $\bar{s}^2$ $\bar{d}_0^2$ $\bar{s}^3$ $\bar{d}_0^3$ $\bar{s}^4$ $\bar{d}_0^4$ $\bar{s}^5$ $\bar{d}_0^5$
s^1 d_0^1 s^2 d_0^2 s^3 d_0^3 s^4 d_0^4 s^5 d_0^5
$\bar{s}^1$

Figure 4.2 Our proposed model. The embedding layer maps the inputs to a high-dimensional vector space. On the right, an RNN decoder stores the information of the decoded sequence. Then, the RNN hidden state and embedded input produce a probability distribution over the next input using the attention mechanism.

As illustrated in Figure 4.2, our model is composed of two main components. The first is a set of graph embeddings [74] that can be used to encode structured data inputs. Among the available techniques, we tried a one-layer Graph Convolutional Network [46] embedding, but it did not show any improvement on the results, so we kept the embedding in this chapter simple by utilizing the local information at each node, e.g., its coordinates and demand values, without incorporating adjacency information. In fact, this embedding maps each customer's information into a D-dimensional vector space encoding. We might have multiple embeddings corresponding to different elements of the input, but they are shared among the inputs. The second component is a decoder that points to an input at every decoding step. As is common in the literature [7, 20, 85], we use RNN to model the decoder network. Notice that we feed the static elements as the inputs to the decoder network. The dynamic elements can also be an input to the decoder, but our experiments on the VRP do not suggest any improvement by doing so. For this reason, the dynamic elements are used only in the attention layer, described next.

4.3.3 Attention Mechanism

An attention mechanism is a differentiable structure for addressing different parts of the input. Figure 4.2 illustrates the attention mechanism employed in our method. At decoder step i, we utilize a context-based attention mechanism with glimpse, similar to Vinyals et al. [97], which extracts the relevant information from the inputs using a variable-length alignment vector a_t. In other words, a_t specifies how much every input data point might be relevant in the next decoding step t.

Let $\bar{x}_t^i = (\bar{s}^i, \bar{d}_t^i)$ be the embedded input i, and $h_t \in \mathbb{R}^D$ be the memory state of the RNN cell at decoding step t. The alignment vector a_t is then computed as

$$a_t = a_t(\bar{x}_t, h_t) = \operatorname{softmax}(u_t), \tag{4.4}$$

where

$$u_t^i = v_a^T \tanh\left(W_a[\bar{x}_t^i; h_t]\right). \tag{4.5}$$

Here ";" means the concatenation of two vectors. We compute the conditional probabilities by combining the context vector c_t, computed as

$$c_t = \sum_{i=1}^{M} a_t^i \bar{x}_t^i, \tag{4.6}$$

with the embedded inputs, and then normalizing the values with the softmax function, as follows:

$$\pi(\cdot|Y_t, X_t) = \operatorname{softmax}(\tilde{u}_t), . \tag{4.7}$$

where

$$\pi(\cdot|Y_t, X_t) = \tilde{u}_t^i = v_c^T \tanh\left(W_c[\bar{x}_t^i; c_t]\right). \tag{4.8}$$

In (4.4)–(4.8), v_a, v_c, W_a and W_c are trainable variables.

Remark 2: *Model Symmetry*: Vinyals et al. [97] discuss an extension of sequence-to-sequence models where they empirically demonstrate that in tasks with no obvious input sequence, such as sorting, the order in which the inputs are fed into the network matters. A similar concern arises when using Pointer Networks for combinatorial optimization problems. However, the policy model proposed in this chapter does not suffer from such a complication since the embeddings and the attention mechanism are invariant to the input order.

4.3.4 Training Methods

To train the network, we use well-known policy gradient approaches. To use these methods, we parameterize the stochastic policy π with parameters θ, where θ is vector of all trainable variables used in the embedding, decoder, and attention mechanism. Policy gradient methods use an estimate of the gradient of the expected return with respect to the policy parameters to iteratively improve the policy. In principle, the policy gradient algorithm contains two networks: (i) an actor network that predicts a probability distribution over the next action at any given decision step, and (ii) a critic network that estimates the reward for any problem instance from a given state.

Let us consider a family of problems, denoted by $\mathcal{M}$, and a probability distribution over them, denoted by $\Phi_{\mathcal{M}}$. During the training, the problem instances are generated according to distribution $\Phi_{\mathcal{M}}$. We also use the same distribution in the inference to produce test examples.

We utilize the REINFORCE method, similar to Bello et al. [8] for solving the TSP and VRP, and A3C [62] for the SVRP. Next, we explain the details of the algorithms.

REINFORCE Algorithm for VRP Algorithm 4 summarizes the REINFORCE algorithm. We have two neural networks with weight vectors θ and ϕ associated with the actor and critic, respectively. We draw N sample problems from $\mathcal{M}$ and use Monte Carlo simulation to produce feasible sequences with respect to the current policy π_θ. We adopt the superscript n to refer to the variables of the nth instance. After termination of the decoding in all N problems, we compute the corresponding rewards as well as the policy gradient in step 14 to update the actor network. In this step, $V(X_0^n; \phi)$ is the the reward approximation for instance problem n that will be calculated from the critic network. We also update the critic network in step 15 in the direction of reducing the difference between the expected rewards with the observed ones during Monte Carlo roll-outs.

Algorithm 4 REINFORCE Algorithm

```
1: initialize the actor network with random weights $\theta$ and critic network with random
   weights $\phi$
2: for $iteration = 1, 2, \cdots$ do
3:     reset gradients: $d\theta \leftarrow 0$, $d\phi \leftarrow 0$
4:     sample $N$ instances according to $\Phi_{\mathcal{M}}$
5:     for $n = 1, \cdots, N$ do
6:         initialize step counter $t \leftarrow 0$
7:         repeat
8:             choose $y_{t+1}^n$ according to the distribution $\pi(\cdot|Y_t^n, X_t^n; \theta)$
9:             observe new state $X_{t+1}^n$
10:            $t \leftarrow t + 1$
11:        until termination condition is satisfied
12:        compute reward $R^n = R(Y^n, X_0^n)$
13:    end for
14:    $d\theta \leftarrow \frac{1}{N} \sum_{n=1}^{N} (R^n - V(X_0^n; \phi)) \nabla_\theta \log P(Y^n|X_0^n; \theta)$
15:    $d\phi \leftarrow \frac{1}{N} \sum_{n=1}^{N} \nabla_\phi (R^n - V(X_0^n; \phi))^2$
16:    update $\theta$ using $d\theta$ and $\phi$ using $d\phi$.
17: end for
```

Asynchronous Advantage Actor-Critic for SVRP The *Asynchronous Advantage Actor-Critic* (A3C) method proposed in [62] is a policy gradient approach that has been shown to achieve super-human performance playing Atari games. In this chapter, we utilize this algorithm for training the policy in the SVRP. In this architecture, we have a central network with weights θ^0, ϕ^0 associated with the actor and critic, respectively. In addition, P agents are running in parallel threads, each having their own set of local network parameters; we denote by θ^p, ϕ^p the actor and critic weights of thread p. (We will use superscript p to denote the operations running on thread p.) Each agent interacts with its own copy of the VRP at the same time as the other agents are interacting with theirs; at each time-step, the vehicle chooses the next point to visit and receives some reward (or cost) and then goes to the next time-step. In the SVRP that we consider in this chapter, R_t is the number of demands satisfied at time t. We note that the system is basically a continuous-time MDP, but in this algorithm, we consider it as a discrete-time MDP running on the times of system state changes $\{\tau_t : t = 0, \cdots\}$; for this reason, we normalize the reward R_t with the duration from the previous time step, e.g., the reward is $R_t/(\tau_t - \tau_{t-1})$. The goal of each agent is to gather independent experiences from the other agents and send the gradient updates to the central network located in the main thread. In this approach, we periodically update

Algorithm 5 Asynchronous Advantage Actor-Critic (A3C)

```
1: initialize the actor network with random weights $\theta^0$ and critic network with random weights $\phi^0$ in the master thread
2: initialize $P$ thread-specific actor and critic networks with weights $\theta^p$ and $\phi^p$ associated with thread $p$
3: repeat
4:   for each thread $p$ do
5:     sample a instance problem from $\Phi_{\mathcal{M}}$ with initial state $X_0^p$
6:     initialize step counter $t^p \leftarrow 0$
7:     while episode not finished do
8:       choose $y_{t+1}^p$ according to $\pi(\cdot|Y_t^p, X_t^p; \theta^p)$
9:       observe new state $X_{t+1}^p$
10:      observe one-step reward $R_t^p = R(Y_t^p, X_t^p)$
11:      let $A_t^p = \left(R_t^p + V(X_{t+1}^p; \phi^p) - V(X_t^p; \phi^p)\right)$
12:      $d\theta^0 \leftarrow d\theta^0 + A_t^p \nabla_\theta \log \pi(y_{t+1}^p|Y_t^p, X_t^p; \theta^p)$
13:      $d\phi^0 \leftarrow d\phi^0 + \nabla_\phi \left(A_t^p\right)^2$
14:      $t^p \leftarrow t^p + 1$
15:    end while
16:  end for
17:  periodically update $\theta^0$ using $d\theta^0$ and $\phi^0$ using $d\phi^0$
18:  $\theta^p \leftarrow \theta^0$, $\phi^p \leftarrow \phi^0$
19:  reset gradients: $d\theta^0 \leftarrow 0$, $d\phi^0 \leftarrow 0$
20: until training is finished
```

the central network weights by accumulated gradients and send the updated weight to all threads. This asynchronous update procedure leads to a smooth training since the gradients are calculated from independent VRP instances.

Both actor and critic networks in this experiment are exactly the same as the ones that we employed for the classical VRP. For training the central network, we use RMSProp optimizer with learning rate 10^{-5}.

4.4 Computational Experiments

In the first experiment of this section, we use the *Traveling Salesman Problem* (TSP) (a special case of the VRP in which there is only a single route to optimize) as the test-bed to validate the performance of the proposed method. In the next experiments, we illustrate the effectiveness of the proposed method in multiple VRP settings. Many variants of the VRP have been extensively studied in the operations research literature. See, for example, the reviews by Laporte [49] or Laporte et al. [50] or the book by Toth and Vigo [94], for different variants of the problem. In this chapter, we study three different VRPs: *i*) Capacitated VRP, *ii*) Capacitated VRP with split delivery, and *iii*) stochastic VRP.

4.4.1 Our Policy Model versus Pointer Network for TSP

We compare the route lengths of the TSP solutions obtained by our framework with those given by the model of Bello et al. [8] for random instances with 20, 50, and 100 nodes. In the training phase, we generate 10^6 TSP instances for each problem size, and use them in training for 20 epochs. We choose 10^6 because we want to have a diverse set of problem configurations; it can be larger or smaller, or we can generate the instances on-the-fly as long as we make sure that the instances are drawn from the same probability distribution with the same random seed. The city locations are chosen uniformly from the unit square $[0, 1] \times [0, 1]$. We use the same data distribution to produce instances for the testing phase. The decoding process starts from a random TSP node and the termination criterion is that all cities are visited. We also use a masking scheme to prohibit visiting nodes more than once.

Table 4.1 summarizes the results for different TSP sizes using the *greedy decoder* in which at every decoding step, the city with the highest probability is chosen as the destination. The results are averaged over 1000 instances. The first column is the average TSP tour length using our proposed architecture, the second column is the result of our implementation of Bello et al. [8] with greedy decoder, and the optimal tour lengths are reported in the last column. To obtain the optimal values, we solved the TSP using the Concorde optimization software [5]. A comparison of the first two columns suggests that there is almost no difference between the performance of our framework and Pointer-RL. In fact, the RNN encoder of the Pointer Network learns to convey no information to the next steps, i.e., $h_t = f(x_t)$. On the other hand, our approach is around 60% faster in both training and inference, since it has two fewer RNNs—one in the encoder of the actor network and another in the encoder of the critic network. Table 4.1 also summarizes the training times for one epoch of the training and the time-savings that we gain by eliminating the encoder RNNs.

Based on the discussion in Section 4.3.1, the main problem with applying Pointer Net-

Table 4.1 Average tour length for TSP and training time for one epoch (in minutes).

	Average tour length			Training time		
Task	Our Framework (Greedy)	Pointer-RL (Greedy)	Optimal	Our Framework (Greedy)	Pointer-RL (Greedy)	% Time Saving
TSP20	3.97	3.96	3.84	22.18	50.33	55.9%
TSP50	6.08	6.05	5.70	54.10	147.25	63.3%
TSP100	8.44	8.45	7.77	122.10	300.73	59.4%

works is mainly computational, and in the next experiment of this section, we compare the learning process of our model with that of Pointer Networks. We implemented a Pointer Network for VRP10, and as is illustrated in Figure 4.3, its performance is much worse, and each training epoch takes around 2.5 times longer to train.

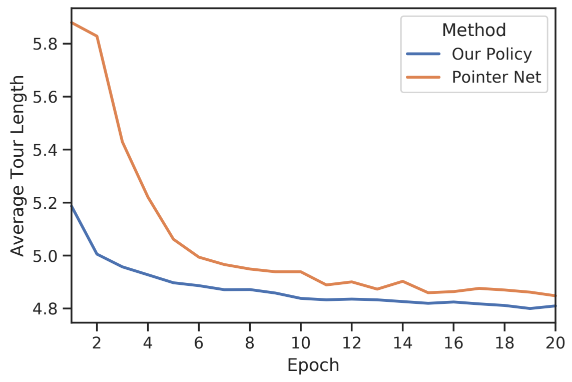


Figure 4.3 Comparison with Pointer Network for VRP.

4.4.2 Capacitated VRP

In this section, we consider a specific capacitated version of the problem in which one vehicle with a limited capacity is responsible for delivering items to many geographically distributed customers with finite demands. When the vehicle's load runs out, it returns to the depot to refill. We will denote the vehicle's remaining load at time t as l_t. The objective is to minimize the total route length while satisfying all of the customer demands. This problem is often called the capacitated VRP (CVRP) to distinguish it from other variants, but we will refer to it simply as the VRP.

We assume that the node locations and demands are randomly generated from a fixed distribution. Specifically, the customers and depot locations are randomly generated in the unit square $[0,1] \times [0,1]$. For simplicity of exposition, we assume that the demand of each node is a discrete number in $\{1,..,9\}$, chosen uniformly at random. We note, however, that the demand values can be generated from any distribution, including continuous ones.

We assume that the vehicle is located at the depot at time 0, so the first input to the decoder is an embedding of the depot location. At each decoding step, the vehicle chooses from among the customer nodes or the depot to visit in the next step. After visiting customer node i, the demands and vehicle load are updated as follows:

$$d_{t+1}^{i} = \max(0, d_t^i - l_t), \quad d_{t+1}^{k} = d_t^k \text{ for } k \neq i, \text{and} \quad l_{t+1} = \max(0, l_t - d_t^i) \tag{4.9}$$

which is an explicit definition of the state transition function (4.2) for the VRP. Once a sequence of the nodes to be visited is sampled, we compute the total vehicle distance and use its negative value as the reward signal.

In this experiment, we have employed two different decoders: *(i)* greedy, in which at every decoding step, the node (either customer or depot) with the highest probability is selected as the next destination, and *(ii)* beam search (BS), which keeps track of the most probable paths and then chooses the one with the minimum tour length [66]. Our results indicate that by applying the beam search algorithm, the quality of the solutions can be improved with only a slight increase in computation time.

For faster training and generating feasible solutions, we have used a *masking scheme* which sets the log-probabilities of infeasible solutions to $-\infty$ or forces a solution if a particular condition is satisfied. In the VRP, we use the following masking procedures: *(i)* nodes with zero demand are not allowed to be visited; *(ii)* all customer nodes will be masked if the vehicle's remaining load is exactly 0; and *(iii)* the customers whose demands are greater than the current vehicle load are masked. Notice that under this masking scheme, the vehicle must satisfy all of a customer's demands when visiting it. We note, however, that if the situation being modeled does allow split deliveries, one can relax *(iii)*. Indeed, the relaxed masking allows split deliveries, so the solution can allocate the demands of a given customer into multiple routes. This property is, in fact, an appealing behavior that is present in many real-world applications, but is often neglected in classical VRP algorithms.

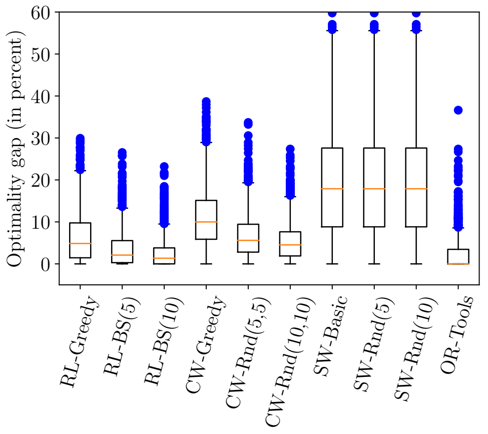
In all the experiments of this section, we do not allow split demands. Further investigation and illustrations of this property are included in Sections 4.4.3 and 4.4.5.

Implementation Details For the embedding, we use 1-dimensional convolution layers for the embedding, in which the in-width is the input length, the number of filters is D, and the number of in-channels is the number of elements of x. We find that training without an embedding layer always yields an inferior solution. One possible explanation is that the policy is able to extract useful features from the high-dimensional input representations much more efficiently. Recall that our embedding is an affine transformation, so it does not necessarily keep the embedded input distances proportional to the original 2-dimensional Euclidean distances.

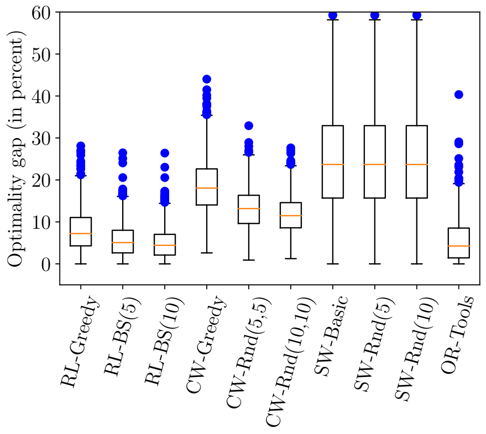
We use one layer of LSTM RNN in the decoder with a state size of 128. Each customer location is also embedded into a vector of size 128, shared among the inputs. We employ similar embeddings for the dynamic elements; the demand d_t^i and the remaining vehicle load after visiting node i, $l_t - d_t^i$, are mapped to a vector in a 128-dimensional vector space and used in the attention layer. In the critic network, first, we use the output probabilities of the actor network to compute a weighted sum of the embedded inputs, and then, it has two hidden layers: one dense layer with ReLU activation and another linear one with a single output. The variables in both actor and critic network are initialized with Xavier initialization [33]. For training both networks, we use the REINFORCE Algorithm and Adam optimizer [45] with learning rate 10^{-4}. The batch size N is 128, and we clip the gradients when their norm is greater than 2. We use dropout with probability 0.1 in the decoder LSTM. Moreover, we tried the entropy regularizer [62, 100], which has been shown to be useful in preventing the algorithm from getting stuck in local optima, but it does not show any improvement in our experiments; therefore, we do not use it in the results reported in this chapter.

On a single GPU K80, every 100 training steps of the VRP with 20 customer nodes takes approximately 35 seconds. Training for 20 epochs requires about 13.5 hours. The TensorFlow implementation of our code is available at `https://github.com/OptMLGroup/VRP-RL`.

Results for Capacitated VRP We compare the solutions found using our framework with those obtained from the *Clarke-Wright savings heuristic* (CW), the *Sweep heuristic* (SW), and Google's optimization tools (OR-Tools). We run our tests on problems with 10, 20, 50 and 100 customer nodes and corresponding vehicle capacity of 20, 30, 40 and 50; for example, VRP10 consists of 10 customers and the default vehicle capacity is 20 unless otherwise specified. The results are based on 1000 instances, sampled for each problem size.



(a) Comparison for VRP10



(b) Comparison for VRP20

	RL-Greedy	RL-BS(5)	RL-BS(10)	CW-Greedy	CW-Rnd(5,5)	CW-Rnd(10,10)	SW-Basic	SW-Rnd(5)	SW-Rnd(10)	OR-Tools
RL-Greedy		12.2	7.2	99.4	97.2	96.3	97.9	97.9	97.9	41.5
RL-BS(5)	85.8		12.5	99.7	99.0	98.7	99.1	99.1	99.1	54.6
RL-BS(10)	91.9	57.7		99.8	99.4	99.2	99.3	99.3	99.3	60.2
CW-Greedy	0.6	0.3	0.2		0.0	0.0	68.9	68.9	68.9	1.0
CW-Rnd(5,5)	2.8	1.0	0.6	92.2		30.4	84.5	84.5	84.5	3.5
CW-Rnd(10,10)	3.7	1.3	0.8	97.5	68.0		86.8	86.8	86.8	4.7
SW-Basic	2.1	0.9	0.7	31.1	15.5	13.2		0.0	0.0	1.4
SW-Rnd(5)	2.1	0.9	0.7	31.1	15.5	13.2	0.0		0.0	1.4
SW-Rnd(10)	2.1	0.9	0.7	31.1	15.5	13.2	0.0	0.0		1.4
OR-Tools	58.5	45.4	39.8	99.0	96.5	95.3	98.6	98.6	98.6	

(c) Comparison for VRP50

	RL-Greedy	RL-BS(5)	RL-BS(10)	CW-Greedy	CW-Rnd(5,5)	CW-Rnd(10,10)	SW-Basic	SW-Rnd(5)	SW-Rnd(10)	OR-Tools
RL-Greedy		25.4	20.8	99.9	99.8	99.7	99.5	99.5	99.5	44.4
RL-BS(5)	74.4		35.3	100.0	100.0	99.9	100.0	100.0	100.0	56.6
RL-BS(10)	79.2	61.6		100.0	100.0	100.0	99.8	99.8	99.8	62.2
CW-Greedy	0.1	0.0	0.0		0.0	0.0	65.2	65.2	65.2	0.0
CW-Rnd(5,5)	0.2	0.0	0.0	92.6		32.7	82.0	82.0	82.0	0.7
CW-Rnd(10,10)	0.3	0.1	0.0	97.2	65.8		85.4	85.4	85.4	0.8
SW-Basic	0.5	0.0	0.2	34.8	18.0	14.6		0.0	0.0	0.0
SW-Rnd(5)	0.5	0.0	0.2	34.8	18.0	14.6	0.0		0.0	0.0
SW-Rnd(10)	0.5	0.0	0.2	34.8	18.0	14.6	0.0	0.0		0.0
OR-Tools	55.6	43.4	37.8	100.0	99.3	99.2	100.0	100.0	100.0	

(d) Comparison for VRP100

Figure 4.4 Parts 4.4a and 4.4b show the optimality gap (in percent) using different algorithms/solvers for VRP10 and VRP20. Parts 4.4c and 4.4d give the proportion of the samples for which the algorithms in the rows outperform those in the columns; for example, RL-BS(5) is superior to RL-greedy in 85.8% of the VRP50 instances.

Figure 4.4 shows the distribution of total solution lengths generated by our method, using greedy and BS decoders, with the number inside the parentheses indicating the beam-width parameter. In the experiments, we label our method with the "RL" prefix. In addition, we also implemented a randomized version of both heuristic algorithms to improve the solution quality; for Clarke-Wright, the numbers inside the parentheses are the randomization depth and randomization iterations parameters; and for Sweep, it is the number of random initial angles for grouping the nodes. Finally, we use Google's OR-Tools [36], which is a more competitive baseline. See Appendix 4.A for a detailed discussion of the baselines.

For small problems of VRP10 and VRP20, it is possible to find the optimal solution, which we do by solving a mixed integer formulation of the VRP [94]. Figures 4.4a and 4.4b measure how far the solutions are far from optimality. The optimality gap is defined as the distance from the objective value of a given solution to the optimal objective value, normalized by the latter. We observe that using a beam width of 10 is the best-performing method; roughly 95% of the instances are at most 10% away from optimality for VRP10 and 13% for VRP20. Even the outliers are within 20–25% of optimality, suggesting that our RL-BS methods are robust in comparison to the other baseline approaches.

Since obtaining the optimal objective values for VRP50 and VRP100 is not computationally affordable, in Figures 4.4d and 4.4d, we compare the algorithms in terms of their winning rate. Each table gives the percentage of the instances in which the algorithms in the rows outperform those in the columns. In other words, the cell corresponding to (A,B) shows the percentage of the samples in which algorithm A provides shorter solutions than B. We observe that the classical heuristics are outperformed by the other approaches in almost 100% of the samples. Moreover, RL-greedy is comparable with OR-Tools, but incorporating beam search into our framework increases the winning rate of our approach to above 60%.

Figure 4.5 shows the solution times normalized by the number of customer nodes. We observe that this ratio stays almost the same for RL with different decoders. In contrast, the run time for the Clarke-Wright and Sweep heuristics increases faster than linearly with the number of nodes. This observation is one motivation for applying our framework to more general combinatorial problems, since it suggests that our method scales well. Even though the greedy Clark-Wright and basic Sweep heuristics are fast for small instances, they do not provide competitive solutions. Moreover, for larger problems, our framework is faster than the randomized heuristics. We also include the solution times for OR-Tools in the graph, but we should note that OR-Tools is implemented in C++, which makes exact time comparisons impossible since the other baselines are implemented in Python. It is worthwhile to mention that the runtimes reported for the RL methods are for the case when we decode a single problem at a time. It is also possible to decode all 1000 test problems in a batch, which will result in approximately 50× speed up. For example, one-by-one decoding of VRP10 for 1000 instances takes around 50 seconds, but by passing all 1000 instances to the decoder at once, the total decoding time decreases to around 1 second on a K80 GPU.

Active search is another method used by [8] to assist with the RL training on a specific problem instance in order to iteratively search for a better solution. We do not believe that active search is a practical tool for our problem. One reason is that it is very time-consuming. A second is that we intend to provide a solver that produces solutions by just scoring a trained policy, while active search requires a separate training phase for every instance. To test our conjecture that active search will not be effective for our problem, we implemented active search for VRP10 with samples of size 640 and 1280, and the average solution length was 4.78 and 4.77 with 15s and 31s solution times per instance, which are far worse than BS decoders. Note that BS(5) and BS(10) give 4.72 and 4.68, in less than 0.1s. For this reason, we exclude active search from our comparisons.

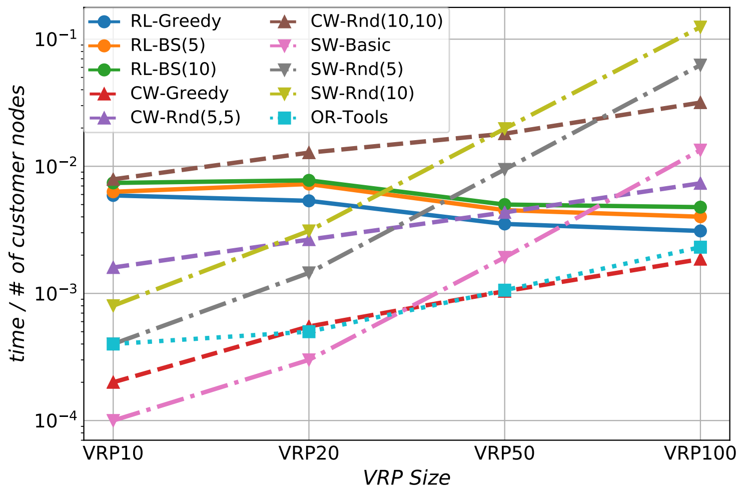


Figure 4.5 Ratio of solution time to the number of customer nodes using different algorithms.

One desired property of the method is that it should be able to handle variable problem

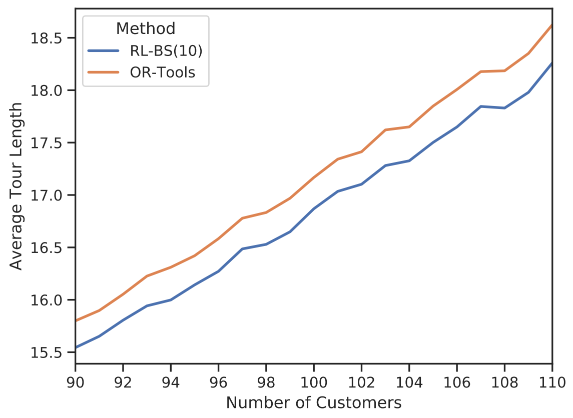


Figure 4.6 Trained for VRP100 and tested for VRP90-VRP110.

sizes. To test this property, we designed two experiments. In the first experiment, we used the trained policy for VRP100 and evaluated its performance on VRP90-VRP110. As can be seen in Figure 4.6, our method with BS(10) outperforms OR-Tools on all problem sizes. In the second experiment, we test the generalization when the problems are very different. More specifically, we use the models trained for VRP50-Cap40 and VRP50-Cap50 in order to generate a solution for VRP100-Cap50. Using BS(10), the average tour length is 18.00 and 17.80, which is still better than the classical heuristics, but worse than OR-Tools. Overall, these two experiments suggest that when the problems are close in terms of the number of customer and vehicle capacity, it is reasonable to expect a high-quality solution, but we will see a degradation when the testing problems are very different from the training ones.s

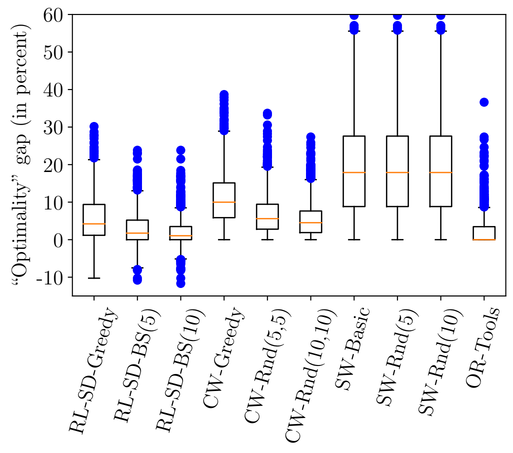
4.4.3 Flexibility to VRPs with Split Deliveries

In the classical VRP that we studied in Section 4.4.2, each customer is required to be visited exactly once. In contrast to what is usually assumed in the classical VRP, one can relax this constraint to obtain savings by allowing split deliveries [6]. In this section, we show that this relaxation is straightforward by slightly modifying the masking scheme. Basically, we omit condition *(iii)* from the masking introduced in Section 4.4.2, and use the new masking with the exact same model; we want to emphasize that we do not re-train the policy model and use the variables trained previously, so this property is achieved at no extra cost.

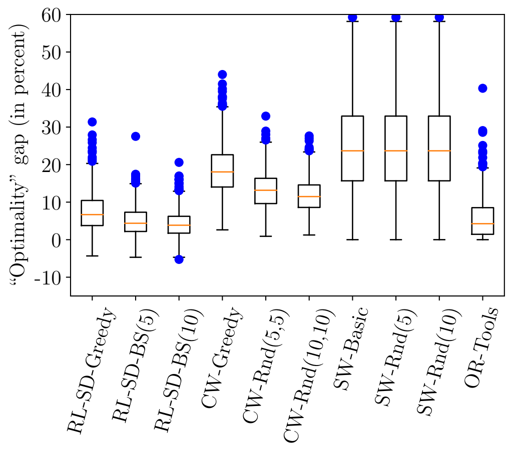
Figure 4.7 shows the improvement by relaxing these constraints, where we label our relaxed method with “RL-SD”. Other heuristics does not have such an option and they are reported for the original (not relaxed) problem. In parts 4.7a and 4.7b we illustrate the “optimality” gap for VRP10 and VRP20, respectively. What we refer to as optimality in this section (and other places in this chapter) is the optimal objective value of the non-relaxed problem. Of course, the relaxed problem may have a lower optimal objective value. That is why RL-SD obtains negative values in these plots. We see that RL-SD can effectively use split delivery to obtain solutions that are around 5%-10% shorter than the “optimal” tours. Similar to Figure 4.4, Figures 4.7c and 4.7d show the winning percentage of the algorithms in rows in comparison to the ones in columns. We observe that the winning percentage of RL-SD methods significantly improves after allowing split demands. For example, in VRP50 and VRP100, RL-SD-Greedy provides competitive results with OR-Tools, or RL-SD-BS(10) outperforms OR-Tools in roughly 67% of the instances, while this number was around 61% before relaxation.

4.4.4 Summary of Comparison with Baselines

Table 4.2 provides the average and the standard deviation of solution lengths for different VRPs. We also test the RL approach using the split delivery option where the customer demands can be satisfied using more than one tour (labeled with “RL-SD”, at the end of the table). We observe that the average total length of the solutions found by our method using various decoders outperforms the heuristic algorithms and OR-Tools. We also see that using the beam search decoder significantly improves the solution while only adding a small computational cost in run-time. Also, allowing split delivery enables our RL-based methods to improve the total tour length by a factor of around 0.6% on average. We also present the solution time comparisons in this table, where all the times are reported on a single core Intel 2.6 GHz CPU. It is worth mentioning that, unlike other RL areas, our findings are not affected by the training seed. This is because during the training, we generate 10^6 problem instances, which is quite adequate to cover various realizations of the problem, and changing the random seed does not significantly change the training and testing instances.



(a) Comparison for VRP10



(b) Comparison for VRP20

	RL-SD-Greedy	RL-SD-BS(5)	RL-SD-BS(10)	CW-Greedy	CW-Rnd(5,5)	CW-Rnd(10,10)	SW-Basic	SW-Rnd(5)	SW-Rnd(10)	OR-Tools
RL-SD-Greedy		15.8	8.7	99.7	98.0	96.9	98.8	98.8	98.8	46.8
RL-SD-BS(5)	82.1		15.4	99.7	99.4	99.3	99.4	99.4	99.4	60.5
RL-SD-BS(10)	90.4	59.2		99.9	99.5	99.6	99.6	99.6	99.6	66.0
CW-Greedy	0.3	0.3	0.1		0.0	0.0	68.9	68.9	68.9	1.0
CW-Rnd(5,5)	2.0	0.6	0.5	92.2		30.4	84.5	84.5	84.5	3.5
CW-Rnd(10,10)	3.1	0.7	0.4	97.5	68.0		86.8	86.8	86.8	4.7
SW-Basic	1.2	0.6	0.4	31.1	15.5	13.2		0.0	0.0	1.4
SW-Rnd(5)	1.2	0.6	0.4	31.1	15.5	13.2	0.0		0.0	1.4
SW-Rnd(10)	1.2	0.6	0.4	31.1	15.5	13.2	0.0	0.0		1.4
OR-Tools	53.2	39.5	34.0	99.0	96.5	95.3	98.6	98.6	98.6	

(c) Comparison for VRP50

	RL-SD-Greedy	RL-SD-BS(5)	RL-SD-BS(10)	CW-Greedy	CW-Rnd(5,5)	CW-Rnd(10,10)	SW-Basic	SW-Rnd(5)	SW-Rnd(10)	OR-Tools
RL-SD-Greedy		27.5	20.3	100.0	100.0	100.0	99.9	99.9	99.9	50.9
RL-SD-BS(5)	72.3		36.4	100.0	100.0	100.0	99.9	99.9	99.9	63.5
RL-SD-BS(10)	79.7	61.7		100.0	100.0	100.0	100.0	100.0	100.0	68.6
CW-Greedy	0.0	0.0	0.0		0.0	0.0	65.2	65.2	65.2	0.0
CW-Rnd(5,5)	0.0	0.0	0.0	92.6		32.7	82.0	82.0	82.0	0.7
CW-Rnd(10,10)	0.0	0.0	0.0	97.2	65.8		85.4	85.4	85.4	0.8
SW-Basic	0.1	0.1	0.0	34.8	18.0	14.6		0.0	0.0	0.0
SW-Rnd(5)	0.1	0.1	0.0	34.8	18.0	14.6	0.0		0.0	0.0
SW-Rnd(10)	0.1	0.1	0.0	34.8	18.0	14.6	0.0	0.0		0.0
OR-Tools	49.1	36.5	31.4	100.0	99.3	99.2	100.0	100.0	100.0	

(d) Comparison for VRP100

Figure 4.7 Parts 4.4a and 4.4b show the "optimality" gap (in percent) using different algorithms/solvers for VRP10 and VRP20. Parts 4.4c and 4.4d give the proportion of the samples (in percent) for which the algorithms in the rows outperform those in the columns; for example, RL-BS(5) is provides shorter tours compared to RL-greedy in 82.1% of the VRP50 instances.

4.4.5 Sample VRP Solutions

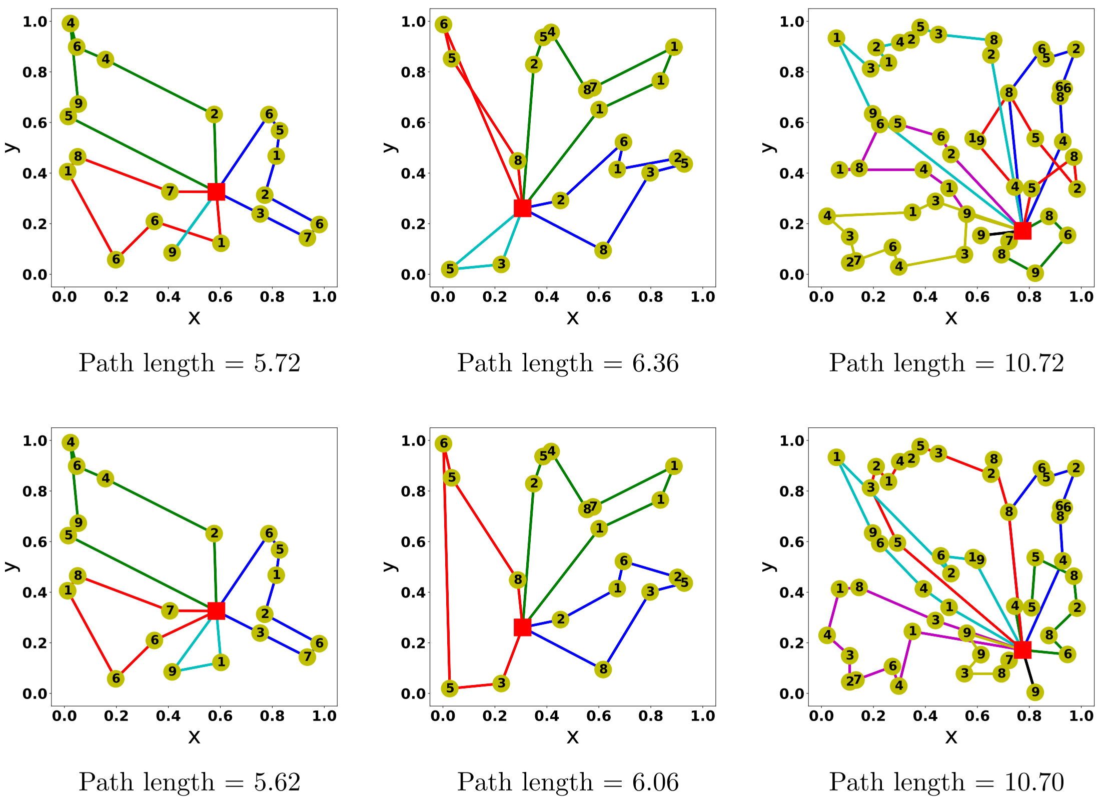
Figure 4.8 illustrates sample VRP20 and VRP50 instances decoded by the trained model. The greedy and beam-search decoders were used to produce the figures in the top and bottom rows, respectively. It is evident that these solutions are not optimal. For example, in part (a), one of the routes crosses itself, which is never optimal in Euclidean VRP instances. Another similar suboptimality is evident in part (c) to make the total distance shorter. However, the figures illustrate how well the policy model has understood the problem structure. It tries to satisfy demands at nearby customer nodes until the vehicle load is small. Then, it automatically comprehends that visiting further nodes is not the best decision, so it returns to the depot and starts a new tour. One interesting behavior

Table 4.2 Average tour length, standard deviations of the tours and the average solution time (in seconds) using different baselines over a test set of size 1000.

Baseline	VRP10, Cap20			VRP20, Cap30			VRP50, Cap40			VRP100, Cap50		
	mean	std	time	mean	std	time	mean	std	time	mean	std	time
RL-Greedy	4.84	0.85	0.049	6.59	0.89	0.105	11.39	1.31	0.156	17.23	1.97	0.321
RL-BS(5)	4.72	0.83	0.061	6.45	0.87	0.135	11.22	1.29	0.208	17.04	1.93	0.390
RL-BS(10)	4.68	0.82	0.072	**6.40**	0.86	0.162	**11.15**	1.28	0.232	**16.96**	1.92	0.445
CW-Greedy	5.06	0.85	0.002	7.22	0.90	0.011	12.85	1.33	0.052	19.72	1.92	0.186
CW-Rnd(5,5)	4.86	0.82	0.016	6.89	0.84	0.053	12.35	1.27	0.217	19.09	1.85	0.735
CW-Rnd(10,10)	4.80	0.82	0.079	6.81	0.82	0.256	12.25	1.25	0.903	18.96	1.85	3.171
SW-Basic	5.42	0.95	0.001	7.59	0.93	0.006	13.61	1.23	0.096	21.01	1.51	1.341
SW-Rnd(5)	5.07	0.87	0.004	7.17	0.85	0.029	13.09	1.12	0.472	20.47	1.41	6.32
SW-Rnd(10)	5.00	0.87	0.008	7.08	0.84	0.062	12.96	1.12	0.988	20.33	1.39	12.443
OR-Tools	**4.67**	0.81	0.004	6.43	0.86	0.010	11.31	1.29	0.053	17.16	1.88	0.231
Optimal	4.55	0.78	0.029	6.10	0.79	102.8		—			—	
RL-SD-Greedy	4.80	0.83	0.059	6.51	0.84	0.107	11.32	1.27	0.176	17.12	1.90	0.310
RL-SD-BS(5)	4.69	0.80	0.063	6.40	0.85	0.145	11.14	1.25	0.226	16.94	1.88	0.401
RL-SD-BS(10)	4.65	0.79	0.074	6.34	0.80	0.155	11.08	1.24	0.250	16.86	1.87	0.477

that the algorithm has learned can be seen in part (c), in which the solution reduces the cost by making a partial delivery; in this example, we observe that the red and blue tours share a customer node with demand 8, each satisfying a portion of its demand; in this way, we are able to meet all demands without needing to initiate a new tour. We also observe how using the beam-search decoder produces further improvements; for example, as seen in parts (b)–(c), it reduces the number of times a tour crosses itself; or it reduces the number of tours required to satisfy all demands as is illustrated in (b).

Tables 4.3 and 4.4 present the RL solutions using the greedy and beam search decoders for two sample VRP10 instances with a vehicle capacity of 20. We have 10 customers indexed $0, \cdots, 9$ and the location with the index 10 corresponds to the depot. The first line specifies the customer locations as well as their demands and the depot location. The solution in the second line is the tour found by the greedy decoder. In the third and fourth line, we observe how increasing the beam width helps in improving the solution quality. Finally, we present the optimal solution in the last row. In Table 4.4, we illustrate an example where our method has discovered a solution by splitting the demands which is, in fact, considerably shorter than the optimal solution found by solving the mixed integer programming model.



(a) Example 1: VRP20; capacity 30

(b) Example 2: VRP20; capacity 30

(c) Example 3: VRP50; capacity 40

Figure 4.8 Illustration of sample decoded solutions for VRP20 and VRP50 using greedy (in top row) and beam-search (bottom row) decoder. The numbers inside the nodes are the demand values.

Table 4.3 Solutions found for a sample VRP10 instance. We use different decoders for producing these solutions; the optimal solution is also presented in the last row.

Sample instance for VRP10:
Customer locations: [[0.411, 0.559], [0.874, 0.302], [0.029, 0.127], [0.188, 0.979], [0.812, 0.330], [0.999, 0.505], [0.926, 0.705], [0.508, 0.739], [0.424, 0.201], [0.314, 0.140]]
Customer demands: [2, 4, 5, 9, 5, 3, 8, 2, 3, 2]
Depot location: [0.890, 0.252]

Greedy decoder:
Tour Length: 5.305
Tour: $10 \rightarrow 5 \rightarrow 6 \rightarrow 4 \rightarrow 1 \rightarrow 10 \rightarrow 7 \rightarrow 3 \rightarrow 0 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 2 \rightarrow 10$

BS decoder with width 5:
Beam tour lengths: [5.305, 5.379, 4.807, 5.018, 4.880]
Best beam: 2, Best tour length: 4.807
Best tour: $10 \rightarrow 5 \rightarrow 6 \rightarrow 4 \rightarrow 1 \rightarrow 10 \rightarrow 7 \rightarrow 3 \rightarrow 0 \rightarrow 10 \rightarrow 8 \rightarrow 2 \rightarrow 9 \rightarrow 10$

BS decoder with width 10:
Beam tour lengths: [5.305, 5.379, 4.807, 5.0184, 4.880, 4.800, 5.091, 4.757, 4.8034, 4.764]
Best beam: 7, Best tour length: 4.757
Best tours: $10 \rightarrow 5 \rightarrow 6 \rightarrow 1 \rightarrow 10 \rightarrow 7 \rightarrow 3 \rightarrow 0 \rightarrow 4 \rightarrow 10 \rightarrow 8 \rightarrow 2 \rightarrow 9 \rightarrow 10$

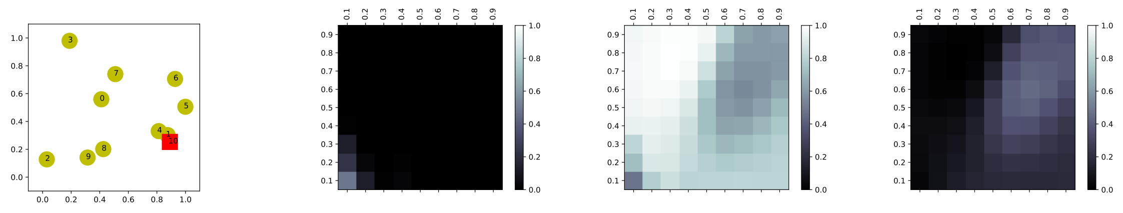
Optimal:
Optimal tour length: 4.546
Optimal tour: $10 \rightarrow 1 \rightarrow 10 \rightarrow 2 \rightarrow 3 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 0 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 10$

4.4.6 Attention Mechanism Visualization

In order to illustrate how the attention mechanism is working, we relocated customer node 0 to different locations and observed how it affects the selected action. Figure 4.9 illustrates the attention in the initial decoding step for a VRP10 instance drawn in part (a). Specifically, in this experiment, we let the coordinates of node 0 equal $\{0.1 \times (i, j), \forall i, j \in \{1, \cdots, 9\}\}$. In parts (b)-(d), the small bottom left square corresponds to the case where node 0 is located at (0.1,0.1) and the others have a similar interpretation. Each small square is associated with a color ranging from black to white, representing the probability of selecting the corresponding node at the initial decoding step. In part (b), we observe that if we relocate node 0 to the bottom-left of the plane, there is a positive probability of directly going to this node; otherwise, as seen in parts (c) and (d), either node 2 or 9 will be chosen with high probability. We do not display the probabilities of the other points since there is a near-0 probability of choosing them, irrespective of the location of node 0. A video demonstration of the decoding process and attention mechanism is available online at `https://youtu.be/qGKt0bB01p0`.

Table 4.4 Solutions found for a sample VRP10 instance where by splitting the demands, our method significantly improves upon the "optimal" (in which no split delivery is allowed).

Sample instance for VRP10: Customer locations: [[0.253, 0.720], [0.289, 0.725], [0.132, 0.131], [0.050, 0.609], [0.780, 0.549], [0.014, 0.920], [0.624, 0.655], [0.707, 0.311], [0.396, 0.749], [0.468, 0.579]] Customer demands: [5, 6, 3, 1, 9, 8, 9, 8, 7, 7] Depot location: [0.204, 0.091]
Greedy decoder: Tour Length: 5.420 Tour: 10 → 7 → 4 → 9 → 10 → 6 → 9 → 8 → 10 → 1 → 0 → 5 → 3 → 10 → 2 → 10
BS decoder with width 5: Beam tour lengths: [5.697, 5.731, 5.420, 5.386, 5.582] Best beam: 3, Best tour length: 5.386 Best tour: 10 → 7 → 4 → 6 → 10 → 6 → 8 → 9 → 10 → 1 → 0 → 5 → 3 → 10 → 2 → 10
BS decoder with width 10: Beam tour lengths: [5.697, 5.731, 5.420, 5.386, 5.362, 5.694, 5.582, 5.444, 5.333, 5.650] Best beam: 8 , Best tour length: 5.333 Best tours: 10 → 7 → 4 → 9 → 10 → 9 → 6 → 8 → 10 → 1 → 0 → 5 → 3 → 10 → 2 → 10
Optimal: Optimal tour length: 6.037 Optimal tour: 10 → 5 → 7 → 10 → 9 → 10 → 2 → 10 → 8 → 10 → 1 → 3 → 4 → 6 → 10



(a) VRP10 instance. **(b)** Point 0. **(c)** Point 2. **(d)** Point 9.

Figure 4.9 Illustration of attention mechanism at decoding step 0. The problem instance is illustrated in part (a) where the nodes are labeled with a sequential number; labels 0-9 are the customer nodes and 10 is the depot. We place node 0 at different locations and observe how it affects the probability distribution of choosing the first action, as illustrated in parts (b)–(d).

4.4.7 Extension to Other VRPs

The proposed framework can be extended easily to problems with multiple depots; one only needs to construct the corresponding state transition function and masking procedure. It is also possible to include various side constraints: soft constraints can be applied by penalizing the rewards, or hard constraints such as time windows can be enforced through a masking scheme. However, designing such a scheme might be a challenging task, possibly harder than solving the optimization problem itself. Another interesting extension is for VRPs with multiple vehicles. In the simplest case in which the vehicles travel independently, one must only design a shared masking scheme to avoid the vehicles pointing to the same customer nodes. Incorporating competition or collaboration among the vehicles is also an interesting line of research that relates to multi-agent RL (MARL) [16].

This framework can also be applied to real-time services including on-demand deliveries and taxis. Next, we design a simulated experiment to illustrate the performance of the framework on the *stochastic VRP* (SVRP), where both customer locations and their demands are subject to change. Our results indicate superior performance compared to the baselines.

Experiment on Stochastic VRP A major difficulty of planning in these systems is that the schedules are not defined beforehand, and one needs to deal with various customer/demand realizations on the fly. Unlike the majority of the previous literature, which only considers one stochastic element (e.g., customer locations are fixed, but the demands can change), we allow the customers and their demands to be stochastic, which makes the problem intractable for many classical algorithms. (See the review of SVRP by Ritzinger et al. [72].) We consider an instance of the SVRP in which customers with random demands arrive at the system according to a Poisson process; without loss of generality we assume the process has rate 1. Similar to previous experiments, we choose each new customer's location uniformly on the unit square and its demand to a discrete number in $\{1, \cdots, 9\}$. We fix the depot position to $(0.5, 0.5)$. A vehicle is required to satisfy as much demand as possible in a time horizon with length 100 time units. To make the system stable, we assume that each customer cancels its demand after having gone unanswered for 5 time units. The vehicle moves with speed 0.1 per time unit. Obviously, this is a continuous-time system, but we view it as a discrete-time MDP where the vehicle can make decisions at either the times of customer arrivals or after the time when the vehicle reaches a node.

The network and its hyper-parameters in this experiment are the same as in the previous experiments. One major difference is the RL training method, where we use asynchronous advantage actor-critic (A3C) [62] with one-step reward accumulation. The main reason for choosing this training method is that REINFORCE is not an efficient algorithm in dealing with the long trajectories. The details of the training method are described in Section 4.3.4. The other difference is that instead of using masking, at every time step, the input to the network is a set of available locations which consists of the customers with positive demand, the depot, and the vehicle's current location; the latter decision allows the vehicle to stop at its current position, if necessary. We also add the *time-in-system* of customers as a dynamic element to the attention mechanism; it will allow the training process to learn customer abandonment behavior.

We compare our results with three other strategies: *(i) Random*, in which the next destination is randomly selected from the available nodes and is meant to provide a "lower bound" on the performance; *(ii) Largest-Demand*, in which the customer with maximum demand will be chosen as the next destination; and *(iii) Max-Reachable*, in which the vehicle chooses the node with the highest demand while making sure that the demand will remain valid until the vehicle reaches the node. In all strategies, we force the vehicle to return to the depot and refill when its load is zero. Even though simple, these baselines are common in many applications. Implementing and comparing the results with more intricate SVRP baselines is an important future direction.

Table 4.5 summarizes the average demand satisfied, and the percentage of the total demand that this represents, under the various strategies, averaged over 100 test instances. We observe that A3C outperforms the other strategies. Even though A3C does not know any information about the problem structure, it is able to perform better than the Max-Reachable strategy, which uses customer abandonment information.

Table 4.5 Satisfied demand under different strategies.

Method	Random	Largest-Demand	Max-Reachable	A3C
Avg. Dem.	24.83	75.11	88.60	112.21
% satisfied	5.4%	16.6%	19.6%	28.8%

4.5 Discussion and Conclusion

According to the findings of this chapter, our RL algorithm is competitive with state-of-the-art VRP heuristics, and this represents progress toward solving the VRP with RL for real applications. The fact that we can solve similar-sized instances without retraining for every new instance makes it easy to deploy our method in practice. For example, a vehicle equipped with a processor can use the trained model and solve its own VRP, only by doing a sequence of pre-defined arithmetic operations. Moreover, unlike many classical heuristics, our proposed method scales well as the problem size increases, and it has superior performance with competitive solution-time. It does not require a distance matrix calculation, which might be computationally cumbersome, especially in dynamically changing VRPs. One important discrepancy which is usually neglected by classical heuristics is that one or more of the elements of the VRP are stochastic in the real world. In this chapter, we also illustrate that the proposed RL-based method can be applied to a more complicated stochastic version of the VRP. In summary, we expect that the proposed architecture has significant potential to be used in real-world problems with further improvements and extensions that incorporate other realistic constraints.

Noting that the proposed algorithm is not limited to VRP, it will be an important topic of future research to apply it to other combinatorial optimization problems such as bin-packing and job-shop or flow-shop scheduling. This method is quite appealing since the only requirement is a verifier to find feasible solutions and also a reward signal to demonstrate how well the policy is working. Once the trained policy is available, it can be used many times, without needing to re-train for new problems as long as they are generated from the training distribution.

Appendix

4.A Capacitated VRP Baselines

In this Appendix, we briefly describe the algorithms and solvers that we used as benchmarks. More details and examples of these algorithms can be found in Snyder and Shen [81]. The first two baseline approaches are well-known heuristics designed for VRP. Our third baseline is Google's optimization tools, which includes one of the best open-source VRP solvers. Finally, we compute the optimal solutions for small VRP instances, so we can measure how far the solutions are from optimality.

4.A.1 Clarke-Wright Savings Heuristic

The Clarke-Wright savings heuristic [24] is one of the best-known heuristics for the VRP. Let $\mathcal{N} \doteq \{1, \cdots, N\}$ be the set of customer nodes, and 0 be the depot. The distance between nodes i and j is denoted by c_{ij}, and c_{0i} is the distance of customer i from the depot. Algorithm 6 describes a randomized version of the heuristic. The basic idea behind this algorithm is that it initially considers a separate route for each customer node i, and then reduces the total cost by iteratively merging the routes. Merging two routes by adding the edge (i, j) reduces the total distance by $s_{ij} = c_{i0} + c_{0j} - c_{ij}$, so the algorithm prefers mergers with the highest savings s_{ij}.

We introduce two hyper-parameters, R and M, which we refer to as the *randomization depth* and *randomization iteration*, respectively. When $M = R = 1$, this algorithm is equivalent to the original Clarke-Wright savings heuristic, in which case, the feasible merger with the highest savings will be selected. By allowing $M, R > 1$, we introduce randomization, which can improve the performance of the algorithm further. In particular, Algorithm 6 chooses randomly from the $r \in \{1, \cdots, R\}$ best feasible mergers. Then, for each r, it solves the problem $m \in \{1, \cdots, M\}$ times, and returns the solution with the shortest total distance.

Algorithm 6 Randomized Clarke-Wright Savings Heuristic

```
1: compute savings $s_{ij}$, where
       $s_{ij} = c_{i0} + c_{0j} - c_{ij}$        $i, j \in \mathcal{N}$, $i \neq j$
       $s_{ii} = 0$                               $i \in \mathcal{N}$
2: for $r = 1, \cdots, R$ do
3:   for $m = 1, \cdots, M$ do
4:     place each $i \in \mathcal{N}$ in its own route
5:     repeat
6:       find $k$ feasible mergers $(i, j)$ with the highest $s_{ij} > 0$, satisfying the following
         conditions:
           i) $i$ and $j$ are in different routes
           ii) both $i$ and $j$ are adjacent to the depot
           iii) combined demand of routes containing $i$ and $j$ is $\leq$ vehicle capacity
7:       choose a random $(i, j)$ from the feasible mergers, and combine the associated
         routes by replacing $(i, 0)$ and $(0, j)$ with $(i, j)$
8:     until no feasible merger is left
9:   end for
10: end for
11: Return: route with the shortest length
```

4.A.2 Sweep Heuristic

The sweep heuristic [101] solves the VRP by breaking it into multiple TSPs. By rotating an arc emanating from the depot, it groups the nodes into several clusters, while ensuring that the total demand of each cluster does not violate the vehicle capacity. Each cluster corresponds to a TSP that can be solved by using an exact or approximate algorithm. In our experiments, we use dynamic programming to find the optimal TSP tour. After solving TSPs, the VRP solution can be obtained by combining the TSP tours. Algorithm 7 shows the pseudo-code of this algorithm.

Algorithm 7 Randomized Sweep Algorithm

```
1: for each $i \in \mathcal{N}$, compute angle $\alpha_i$, respective to depot location
2: $l \leftarrow$ vehicle capacity
3: **for** $r = 1, \cdots, R$ **do**
4:     select a random angle $\alpha$
5:     $k \leftarrow 0$; initialize cluster $S_k \leftarrow \emptyset$
6:     **repeat**
7:         increase $\alpha$ until it equal to some $\alpha_i$
8:         **if** demand $d_i > l$ **then**
9:             $k \leftarrow k + 1$
10:            $S_k \leftarrow \emptyset$
11:            $l \leftarrow$ vehicle capacity
12:        **end if**
13:        $S_k \leftarrow S_k \cup \{i\}$
14:        $l \leftarrow l - d_i$
15:    **until** no unclustered node is left
16:    solve a TSP for each $S_k$
17:    merge TSP tours to produce a VRP route
18: **end for**
19: **Return**: route with the shortest length
```

4.A.3 Google's OR-Tools

Google Optimization Tools (OR-Tools) [36] is an open-source solver for combinatorial optimization problems. OR-Tools contains one of the best available VRP solvers, which has implemented many heuristics (e.g., Clarke-Wright savings heuristic [24], Sweep heuristic [101], Christofides' heuristic [23] and a few others) for finding an initial solution and meta-heuristics (e.g. Guided Local Search [98], Tabu Search [34] and Simulated Annealing [47]) for escaping from local minima in the search for the best solution. The default version of the OR-Tools VRP solver does not exactly match the VRP studied in this chapter, but with a few adjustments, we can use it as our baseline. The first limitation is that OR-Tools only accepts integer locations for the customers and depot while our problem is defined on the unit square. To handle this issue, we scale up the problem by multiplying all locations by 10^4 (meaning that we will have 4 decimal digits of precision), so the redefined problem is now in $[0, 10^4] \times [0, 10^4]$. After solving the problem, we scale down the solutions and tours to get the results for the original problem. The second difference is that OR-Tools is defined for a VRP with multiple vehicles, each of which can have at most one tour. One can verify that by setting a large number of vehicles (10 in our experiments), it is mathematically equivalent to our version of the VRP.

4.A.4 Optimal Solution

We use a mixed integer formulation for the VRP [94] and the Gurobi optimization solver [38] to obtain the optimal VRP tours. VRP has an exponential number of constraints, and of course, it requires careful tricks for even small problems. In our implementation, we start off with a relaxation of the capacity constraints and solve the resulting problem to obtain a lower bound on the optimal objective value. Then we check the generated tours and add the capacity constraint as *lazy-constraints* if a specific subtour's demand has violated the vehicle capacity, or the subtour does not include the depot. With this approach, we were able to find the optimal solutions for VRP10 and VRP20, but this method is intractable for larger VRPs; for example, on a single instance of VRP50, the solution has 6.7% optimality gap after 10000 seconds.

Chapter 5

Don't Forget Your Teacher: A Corrective Reinforcement Learning Framework

5.1 Introduction

We encourage using a new paradigm called *corrective RL*, in which a reinforcement learning (RL) agent is trained to maximize its reward while not straying "too far" from a previously defined policy. The motivation is twofold: (1) to provide a gentler transition for a decision-maker who is accustomed to using a certain policy but now considers implementing RL, and (2) to develop a framework for gently transitioning from one RL solution to another when the underlying environment has changed.

RL has recently achieved considerable success in artificially created environments, such as Atari games [60, 62] or robotic simulators [54]. Exploiting the power of neural networks in RL algorithms has been shown to exhibit super-human performance by enabling automatic feature extraction and policy representations, but real-world applications are still very limited, conceivably due to lack of representativity of the optimized policies. Over the past few years, a major portion of the RL literature has been developed for RL agents with *no prior information* about how to do a task. Typically, these algorithms start with random actions and learn while interacting with the environment through trial and error. However, in many practical settings, prior information about good solutions *is available*, whether from a previous RL algorithm or a human decision-maker's prior experience. Our approach trains the RL agent to make use of this prior information when optimizing, in order to avoid deviating too far from a target policy.

Although toy environments and Atari games are prevalent in the RL literature due to their simplicity, RL has recently been trying to find its path to real-world applications such as recommender systems [19], transportation [65], Internet of Things [28], supply chain [31, 67] and various control tasks in robotics [37]. In all of these applications, there is a crucial risk that the new policy might not operate logically or safely, as one was expecting it to do. A policy that attains a large reward but deviates too much from a known policy—which follows logical steps and processes—is not desirable for these tasks. For example, users of a system who were accustomed to the old way of doing things would likely find it hard to switch to a newly discovered policy, especially if the benefit of the new policy is not obvious or immediately forthcoming. Indeed, we argue that many real-world tasks only need a small *corrective* fix to the currently running policies to achieve their desired goals, instead of designing everything from scratch. Throughout this paper, we adhere to this paradigm—we call it "corrective RL"—which utilizes an acceptable policy as a gauge when designing novel policies. We consider two agents, namely a *teacher* and a *student*. Our main question is how the student can improve upon the teacher's policy while not deviating too far from it. More formally, we would like to train a student in such a way that it maximizes a long-term RL objective while keeping its own policy close to that of the teacher.

For example, consider an airplane that is controlled by an autopilot that follows the shortest haversine path policy towards the destination. Then, some turbulence occurs, and we want to modify the current path to avoid the turbulence. A "pure" RL algorithm would re-optimize the trajectory from scratch, potentially deviating too far from the optimal path in order to avoid the turbulence. Corrective RL would ensure that the adjustments to the current policy are small, so that the flight follows a similar path and has a similar estimated time of arrival, while ensuring that the passengers experience a more comfortable (less turbulent) flight. Another example is in predictive maintenance, where devices are periodically inspected for possible failures. Inspection schedules are usually prescribed by the device designers, but many environmental conditions affect failure rates, hence there is no guarantee that factory schedules are perfect. If the objective is to reduce downtime with only slight adjustments to the current schedules, conventional RL algorithms would have a hard time finding such policies.

Similar concerns arise in other business and engineering domains as well, including supply chain management, queuing systems, finance, and robotics. For example, an enormous number of exact and inexact methods have been proposed for classical inventory management problems under some assumptions on the demand, e.g., that it follows a normal distribution [81]. Once we add more stochasticity to the demand distribution or consider more complicated cost functions, these problems often become intractable using classical methods. Of course, vanilla RL can provide a mechanism for solving more complex inventory optimization problems, but practitioners may prefer policies that are simple to implement and intuitive to understand. Corrective RL can help steer the policy toward the preferred ones, while still maintaining near-optimal performance. Given these examples, one can interpret our approach as an improvement on black-box heuristics, which uses a data-driven approach to improve the performance of these algorithms without dramatic reformulations.

The contributions of this work are as follows: *i*) we introduce a new paradigm for RL tasks, convenient for many real-world tasks, that improves upon the currently running system's policy with a small perturbation of the policy; *ii*) we formulate this problem using a stochastic optimization problem and propose a primal–dual policy gradient algorithm which we prove to be asymptotically optimal, and *iii*) using practical adjustments, we illustrate how an RL framework can act as an improvement heuristic. We show the effectiveness and properties of the algorithm in multiple GridWorld motion planning experiments.

5.2 Problem Definition

We consider the standard definition of a Markov decision process (MDP) using a tuple $(\mathcal{X}, \mathcal{A}, C, P, P_0)$. In our notation, $\mathcal{X} := \mathcal{X}' \cup \{x_{term}\} = \{1, 2, \dots, n, x_{term}\}$ is the state space, where $\mathcal{X}'$ is the set of transient states and x_{term} is the terminal state; $\mathcal{A}$ is the set of actions;

$C : \mathcal{X} \times \mathcal{A} \rightarrow [0, C_{max}]$ is the cost function; P is the transition probability distribution; and P_0 is the distribution of the initial state x_0. At each time step $t = 0, 1, \ldots$, the agent observes $x_t \in \mathcal{X}$, selects $a_t \in \mathcal{A}$, and incurs a cost $c_t = C(x_t, a_t)$. Selecting the action a_t at state x_t transitions the agent to the next state $x_{t+1} \sim P(\cdot|x_t, a_t)$.

Consider two agents, a teacher and a student. The teacher has prior knowledge about the task and prescribes an action for any state that the student encounters, and the student has the authority to follow the teacher's action or act independently based on its own learned policy. Let π_S denote the policy of the student and π_T be the policy of the teacher, where both π_S and π_T are stationary stochastic policies defined as a mapping from a state–action pair to a probability distribution, i.e., $\pi_i : \mathcal{X} \times \mathcal{A} \rightarrow [0, 1]$, $i \in \{S, T\}$. For example, $\pi_S(a|x)$ denotes the probability of choosing action a in state x by the student. In policy gradient methods, the policies are represented with a function approximator, usually modeled by a neural network, where we denote by $\theta \in \mathbb{R}^N$ and $\phi \in \mathbb{R}^N$ the corresponding policy weights of the student and teacher, respectively; the teacher and student parameterized policies are denoted by $\pi_T(\cdot|\cdot; \phi)$ and $\pi_S(\cdot|\cdot; \theta)$. In what follows, we adapt this parameterization structure into the notation and interchangeably refer to π_S and π_T with their associated weights, θ and ϕ.

We consider the simulation optimization setting, where we can sample from the underlying MDP and observe the costs. Consider a possible state–action–cost trajectory τ defined as $\{x_0, a_0, c_0, x_1, a_1, c_1, \cdots, x_{H-1}, a_{H-1}, c_{H-1}, x_H\}$ and let $\mathcal{T} := \{\tau\}$ to be the set of all possible trajectories under all admissible policies. For simplicity of exposition, we assume that the first hitting time H of a terminal state x_{term} from any given x and following a stationary policy is bounded almost surely with an upper bound H_{max}, i.e., $H \leq H_{max}$ almost surely. Since the sample trajectories in many RL tasks terminate in finite time, this assumption is not restrictive. For example, the game fails after reaching a certain state or a time-out signal may terminate the trajectory. Along a trajectory τ, the system incurs a discounted cost $J_\theta(\tau) = \sum_{t=0}^{H-1} \gamma^t c_t$, with discount factor $\gamma \in (0, 1]$, and the probability of sampling such a trajectory is

$$\mathbb{P}_\theta(\tau) = P_0(x_0) \prod_{t=0}^{H-1} \pi_S(a_t|x_t;\theta) P(x_{t+1}|x_t, a_t). \tag{5.1}$$

We denote the expected cost from state x onward until hitting the terminal state x_{term} by $V_\theta(x)$, i.e.,

$$V_\theta(x) = \mathbb{E}_\theta \left[\sum_{t=0}^{H-1} \gamma^t C(x_t, a_t) | x_0 = x \right]. \tag{5.2}$$

5.2.1 Distance Measure

An important question that arises is how to quantify the distance between the policies of the teacher and the student. There are several distance measures studied in the literature for computing the closeness of two probability measures. Among those, the *Kullback-Leibler* (KL) divergence [63] is a widely used metric. In this work, we consider both *reverse* (KL-R) and *forward* (KL-F) KL-divergence, defined as

$$D_{KL}(\theta \parallel \phi) := D_{KL}(\mathbb{P}_\theta(\tau) \parallel \mathbb{P}_\phi(\tau)) = \sum_{\tau \in \mathcal{T}} \mathbb{P}_\theta(\tau) \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)}, \tag{KL-R}$$

$$D_{KL}(\phi \parallel \theta) := D_{KL}(\mathbb{P}_\phi(\tau) \parallel \mathbb{P}_\theta(\tau)) = \sum_{\tau \in \mathcal{T}} \mathbb{P}_\phi(\tau) \log \frac{\mathbb{P}_\phi(\tau)}{\mathbb{P}_\theta(\tau)}. \tag{KL-F}$$

KL-divergence is known to be an asymmetric distance measure, meaning that changing the order of the student and teacher distributions will cause different learning behaviors. We will use the reverse KL-divergence in the theoretical analysis since it provides more compact notation. However, in all of the experiments, we will consider the forward setting, i.e., $D_{KL}(\phi\|\theta)$, unless otherwise specified. Informally speaking, this form of KL-divergence, which is also known as the *mean-seeking* KL, allows the student to perform actions that are not included in the teacher's behavior. This is because *i*) when the teacher can perform an action a_t in a given state s_t, the student should also have $\pi_S(a_t|s_t) > 0$ to keep the distance finite, and *ii*) the student can have $\pi_S(a_t|s_t) > 0$ irrespective of whether the teacher is doing that action or not. The reverse direction, $D_{KL}(\theta\|\phi)$, known as the *mode-seeking* KL, can be useful as well. For example, let's assume that the teacher policy is a mixture of several Gaussian sub-policies. Using the reverse order will allow the student to assign only one sub-policy as its decision making policy. Hence, the choice of reverse KL would be preferred if the student wants to find a policy which is close to a teacher's sub-policy with the highest return. The justification of this behavior is also visible from the definition: when $\pi_S(a_t|s_t) > 0$ for a given state and action, then the teacher also should have $\pi_T(a_t|s_t) > 0$. Also, $\pi_T(a_t|s_t) = 0$ would not allow $\pi_S(a_t|s_t) > 0$. For more detailed discussion and examples, we refer the interested reader to Appendix 5.B.1 and Section 10.1.2 of [63].

5.2.2 Optimization Problems

The student's optimization problems that we would like to solve for reverse KL-divergence (OPT-R) and the forward KL-divergence (OPT-F) are defined as

$$\begin{aligned} &\min_{\theta\in\Theta} V_\theta(x_0) \\ &\text{s.t. } D_{KL}(\theta \parallel \phi) \le \delta \end{aligned} \qquad \text{(OPT-R)} \qquad\qquad \begin{aligned} &\min_{\theta\in\Theta} V_\theta(x_0) \\ &\text{s.t. } D_{KL}(\phi \parallel \theta) \le \delta, \end{aligned} \qquad \text{(OPT-F)}$$

where δ is an upper bound on the KL-divergence and Θ is a convex compact set of possible policy parameters. Most of the theoretical analysis of the two optimization problems is quite similar, so we will use (OPT-R) as our main formulation. In Appendix 5.A.4, we investigate the equivalence of both problems and state their minor differences.

The widely adopted problem studied for MDPs only contains the objective function; however, we impose an additional constraint to restrict the student's policy. By fixing an appropriate value for δ, one can enforce a constraint on the maximum allowed deviation of the student policy from that of the teacher. The objective is to find a set of optimal points θ^* that minimizes the discounted expected cost while not violating the KL constraint. Notice that $\pi_S = \pi_T$ is a trivial feasible solution. In addition, we need to have the following assumption to ensure that (OPT-R) is well-defined:

Assumption 5.2.1. ***(Well-defined*** (OPT-R)***)*** *For any state–action pair* $(x, a) \in \mathcal{X} \times \mathcal{A}$ *with* $\pi_T(x, a) = 0$*, we have* $\pi_S(x, a) = 0$.

Intuitively, Assumption 5.2.1 specifies that when the teacher does not take a specific action in a given state, the student also cannot choose that action. Even though this assumption might seem restrictive, it is valid in situations in which the student is indeed limited to the positive-probability action space of the teacher. Alternatively, we can certify this assumption by adding a small noise term to the outcome of the teacher's policy at the expense of some information loss.

5.2.3 Lagrangian Relaxation of (OPT-R)

The standard method for solving (OPT-R) is by applying Lagrangian relaxation [9]. We define the Lagrangian function

$$L(\theta, \lambda) := V_\theta(x_0) + \lambda \left(D_{\mathrm{KL}}(\theta \parallel \phi) - \delta \right), \tag{5.3}$$

where λ is the Lagrange multiplier. Then the optimization problem (OPT-R) can be converted into the following problem:

$$\max_{\lambda \geq 0} \min_{\theta \in \Theta} \; L(\theta, \lambda). \tag{5.4}$$

The intuition beyond (5.4) is that we now allow the student to deviate arbitrarily much from the teacher's policy in order to decrease the cumulative cost, but we penalize any such deviation.

Next, we define a dynamical system which, as we will prove in Appendix 5.A.2, solves problem (OPT-R) under several common assumptions for stochastic approximation methods. Once we know the optimal Lagrange multiplier λ^*, then the student's optimal policy is

$$\theta^* \in \arg\min_\theta V_\theta(x_0) + \lambda^* \left(D_{\mathrm{KL}}(\theta \parallel \phi) - \delta \right). \tag{5.5}$$

A point (θ^*, λ^*) is a saddle point of $L(\theta, \lambda)$ if for some $r > 0$, we have

$$L(\theta^*, \lambda) \leq L(\theta^*, \lambda^*) \leq L(\theta, \lambda^*) \tag{5.6}$$

for all $\theta \in \Theta \cap \mathbb{B}_r(\theta^*)$ and $\lambda \geq 0$, where $\mathbb{B}_r(\theta^*)$ represents a ball around θ^* with radius r. Then, the saddle point theorem [9] immediately implies that θ^* is the local optimal solution of (OPT-R).

5.3 Primal–Dual Policy Gradient Algorithm

We propose a primal–dual policy gradient (PDPG) algorithm for solving (OPT-R).

5.3.1 PDPG Algorithm

Having derived the gradients of the Lagrangian (in Appendix 5.A.1), we have all the necessary information for proposing our primal–dual policy gradient (PDPG) algorithm, which is described in Algorithm 8.

Algorithm 8 Primal-Dual Policy Gradient (PDPG) Algorithm for (OPT-R)

```
1: input: teacher's policy with weights $\phi$
2: initialize: student's policy with $\theta^0$, possibly equal to $\phi$; initialize step size schedules $\alpha_1(\cdot)$ and $\alpha_2(\cdot)$
3: while TRUE do
4:     for $k = 0, 1, \ldots$ do
5:         following policy $\theta^k$, generate a set of $N$ trajectories $\mathcal{T}^k = \{\tau_j^k, j = 1, 2, \ldots, N\}$, each starting from an initial state $x_0 \sim P_0(\cdot)$
6:         ($\theta$-update) update $\theta^k$ according to
               $$\theta^{k+1} = \Gamma_\Theta\Big[\theta^k - \alpha_1(k)\Big(\frac{1}{N}\sum_{j=1}^{N}\nabla_\theta \log \mathbb{P}_\theta(\tau_j^k)\big|_{\theta=\theta^k}\big(J(\tau_j^k) + \lambda^k \log\frac{\mathbb{P}_\theta(\tau_j^k)}{\mathbb{P}_\phi(\tau_j^k)} + \lambda^k\big)\Big)\Big] \tag{5.7}$$
7:         ($\lambda$-update) update $\lambda^k$ according to
               $$\lambda^{k+1} = \Gamma_\Lambda\Big[\lambda^k + \alpha_2(k)\Big(\frac{1}{N}\sum_{j=1}^{N}\log\frac{\mathbb{P}_\theta(\tau_j^k)}{\mathbb{P}_\phi(\tau_j^k)} - \delta\Big)\Big] \tag{5.8}$$
8:     end for
9:     if $\lambda^k$ converges to $\lambda_{max}$ then
10:        $\lambda_{max} \leftarrow 2\lambda_{max}$
11:    else
12:        return $\theta$ and $\lambda$; break
13:    end if
14: end while
```

After initializing the student, possibly with that of the teacher, we take a mini-batch of sample trajectories under the student's policy θ^k at each iteration k. In step 6, we use the sampled trajectories to compute an approximate gradient of the Lagrangian function with respect to θ and update the policy parameters in the negative direction of the approximate gradient with step size $\alpha_1(k)$. In addition to policy parameter updates, the dual variables are learned concurrently using the recursive formula

$$\lambda^{k+1} = \lambda^k + \alpha_2(k) \left[\hat{D}_{\mathrm{KL}}(\theta \parallel \phi) - \delta \right], \tag{5.9}$$

where $\alpha_2(k)$ represents the associated step-size rule. Finally, using an optimization algorithm, we update θ and λ according to the approximated gradients.

In this algorithm, we need to use two projection operators to ensure the convergence of the algorithm. Specifically, Γ_Θ is an operator that projects θ to the closest point in Θ, i.e., $\Gamma_\Theta(\theta) = \arg\min_{\hat{\theta}\in\Theta} \|\theta - \hat{\theta}\|^2$. Similarly, Γ_Λ is an operator that maps λ to the interval $\Lambda := [0, \lambda_{max}]$. Finally, in steps 9–13, we check whether λ has converged to some point on the boundary. Such a convergence means that the projection space for the Lagrange multipliers is small, so we increment the upper bound and repeat searching for a better policy.

In order to prove our main convergence result, we need some technical assumptions on the student's policy and step sizes.

Assumption 5.3.1. ***(Smooth policy)*** *For any $(x, a) \in \mathcal{X} \times \mathcal{A}$, $\pi_S(a|x;\theta)$ is a continuously differentiable function in θ and its gradient is $\mathscr{L}$-Lipschitz continuous, i.e., for any θ^1 and θ^2,*

$$\left\| \nabla_\theta \pi_S(a|x;\theta)\big|_{\theta=\theta^1} - \nabla_\theta \pi_S(a|x;\theta)\big|_{\theta=\theta^2} \right\| \le \mathscr{L} \|\theta^1 - \theta^2\|. \tag{5.10}$$

Assumption 5.3.2. ***(Step-size rules)*** *The step-sizes $\alpha_1(k)$ and $\alpha_2(k)$ in update rules* (5.7) *and* (5.8) *satisfy the following relations:*

(i) $\sum_k \alpha_1(k) = \infty$; $\sum_k \alpha_1^2(k) < \infty$,

(ii) $\sum_k \alpha_2(k) = \infty$; $\sum_k \alpha_2^2(k) < \infty$,

(iii) $\alpha_2(k) = o(\alpha_1(k))$.

Relations (i) and (ii) in Assumption 5.3.2 are common in stochastic approximation algorithms, and (iii) indicates that the Lagrange multiplier update is in a slower time-scale compared to the policy updates. The latter condition simplifies the convergence proof by allowing us to study the PDPG as a two-time-scale stochastic approximation algorithm. The following theorem states the main theoretical result of this paper.

Theorem 5.3.3. *Under Assumptions 5.2.1, 5.3.1, and 5.3.2, the sequence of policy updates (starting from θ^0 sufficiently close to a local optimum point θ^*) and Lagrange multipliers converges almost surely to a saddle point of the Lagrangian, i.e., $(\theta(k), \lambda(k)) \xrightarrow{a.s.} (\theta^*, \lambda^*)$. Moreover, θ^* is a local optimal solution of* (OPT-R).

Proof. (*sketch*) The proof is similar to those found in [21, 87]. It is based on representing θ and λ update rules with a two-time-scale stochastic approximation algorithm. For each timescale, the algorithm can be shown to converge to the stationary points of the corresponding continuous-time system. Finally, it can be shown that the fixed point is, in fact, a locally optimal point. In Appendix 5.A.2, we provide a formal proof of this theorem. $\square$

Corollary 5.3.4. *Under Assumptions 5.2.1, 5.3.1, and 5.3.2, the sequence of policy updates and Lagrange multipliers converges globally to a stationary point of the Lagrangian almost surely. Moreover, if θ^* is in the interior of Θ, then θ^* is a feasible first order stationary point of* (OPT-R), *i.e.,* $\nabla_\theta V_\theta(x_0)|_{\theta=\theta^*} = 0$ *and* $D_{KL}(\theta^* \parallel \phi) \leq \delta$.

Theorem 5.3.3 and Corollary 5.3.4 are also valid for the forward KL constraint case, as we discuss in Appendix 5.A.4.

5.4 Practical PDPG Algorithm

Although the algorithm presented in the previous section is proved to converge to a first-order stationary point, it cannot directly serve as a practical learner algorithm. The main reason is that it produces a high-variance approximation of the gradients, which would lead to unstable learning. In this section, we propose several approximations to the theoretically-justified PDPG in order to develop a more practical algorithm. For this algorithm, we will consider the forward definition of KL-divergence due to the mean-covering property.

One source of variance is the reward bias, which can be handled by adding a critic, similar to [48]. Our next adjustment is to use an approximation of the step-wise KL-divergence, defined as

$$\hat{D}_{KL}^{step}(\phi \parallel \theta) = \frac{1}{N} \sum_{j=1}^{N} \sum_{t=0}^{H} D_{KL}^{step}\big(\pi_T(\cdot|x_t; \phi) \parallel \pi_S(\cdot|x_t; \theta)\big), \tag{5.11}$$

where

$$D_{KL}^{step}\big(\pi_T(\cdot|x_t; \phi) \parallel \pi_S(\cdot|x_t; \theta)\big) = \sum_{a \in \mathcal{A}} \pi_T(a|x_t; \phi) \log \frac{\pi_T(a|x_t; \phi)}{\pi_S(a|x_t; \theta)}.$$

As we discuss in Appendix 5.B.1, using (5.11) results in a much smaller variance, while still ensuring the convergence results. Intuitively, this equation suggests that instead of computing the trajectory probabilities and then computing the KL-divergence, as in (KL-R), one can compute the KL in every visited state along a trajectory and sum them up. In addition to this change, we will further normalize each D_{KL}^{step} by its trajectory length H to remove the effect of the variable horizon length. The latter modification will lead to more sensible KL values and will make the choice of δ easier.

A second difficulty with the algorithm in Section 5.3 is that, unlike conventional policy gradient algorithms, there is no guarantee that the student's optimal policy is a deterministic one. In fact, in most of our experiments, it happens that the optimal policy is stochastic, especially when the teacher's policy itself is stochastic. To illustrate this, consider two scenarios: *i*) The student refuses to do the suggested action of a deterministic teacher. In this case, she would incur an infinite cost as a result of her disobedience, so the problem will be infeasible. *ii*) The teacher is less informative and has no clue about most of the state space, so often takes random actions. Trying to emulate this teacher would cause degraded performance for the student as well, so the student would also take many less informed actions.

A stochastic optimal policy is usually not desirable since it poses major safety and reliability challenges, so our next adjustments are an attempt to address this issue. One possible mitigation for the first scenario might be using a bounded distance measure such as Hellinger [25] instead of KL-divergence, but our numerical experiments did not confirm that this is effective. We observe that by using the Hellinger constraint, the total entropy of the student's policy stays high, without any improvement in the student's policy. Instead, we propose using *percentile KL-clipping*, which we define as

$$\textsc{clip}_\rho \left(D_{KL}^{step} \right) = \max\{\rho\% \text{ percentile of all } D_{KL}^{step}\text{s at time } t, D_{KL}^{step}\} \tag{5.12}$$

In fact, the CLIP function enables the student to totally disagree with the teacher in $\rho\%$ of the visited states, without receiving an extremely large penalty. Selecting the value for ρ depends on our perception about how perfect the teacher is. Setting ρ close to 100 means that we believe in the teacher's suggestions. As we decrease ρ, we rely less on the teacher and can disobey more freely.

The last major modification is to control the expected entropy at a certain small level $\delta^{ent} > 0$, i.e.,

$$ent(\theta) := -\mathbb{E}_x \left[\frac{1}{H} \sum_{t=0}^{H} \sum_{a\in\mathcal{A}} \pi_S(a|x;\theta) \log \pi_S(a|x;\theta) \right] = \delta^{ent}. \tag{5.13}$$

The justification for adding (5.13) is that we would like the optimal policy to be close to a deterministic one as much as possible. By setting a small value for δ^{ent}, we can enforce this property. Also, this constraint tries to avoid having a deterministic policy in intermediate training steps, in order to allow more exploration. To add this constraint, we use the same Lagrangian technique, adding an extra term to the Lagrangian function:

$$L(\theta, \lambda, \zeta) := V_\theta(x_0) + \lambda \left(D_{\mathrm{KL}}(\phi \parallel \theta) - \delta \right) + \zeta \left(ent(\theta) - \delta^{ent} \right). \tag{5.14}$$

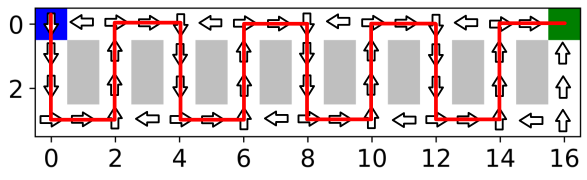
All of these modifications, along with a few others, are summarized in Algorithm 9 of Appendix 5.B.2.

5.5 Experiments

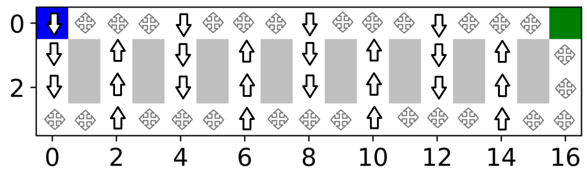
We illustrate the efficiency of the proposed methods with multiple GridWorld experiments. In the first set of experiments, the teacher tries to teach the student to perform an oscillating maneuver around the walls. In the second set, we study how the student can comprehend changes in the environment and utilize them to increase its rewards.

5.5.1 Square-Wave Teacher

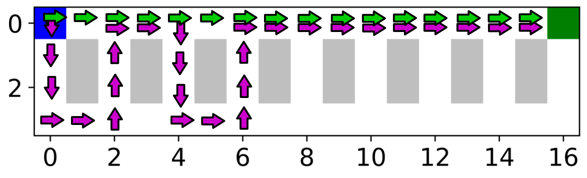
In this experiment, we consider a teacher who gives a suggestion in every state of a Grid-World. We study two variants of the teacher, one who is very determined about all of his suggestions and the other who is less confident. Figure 5.1 illustrates the environment and both teachers' suggestions. A student wants to find a path from the blue state to the green target. Each step has a reward -1 and reaching the target brings $+100$ reward. If the student wants to act independently, the optimal path is a trivial horizontal line. However, our objective is to force the student to "listen" to her teacher up to some level.



(a) "Determined" teacher with the corresponding suggested actions for every state. The red line shows the teacher's suggested path to target.



(b) "Less confident" teacher with deterministic actions in a subset of states and uniformly random actions in the rest.



(c) Optimal Path (in green) versus a sample path found by PDPG (in purple) with $\delta = 0.2$ and $\rho = 8$.

Figure 5.1 Two different teachers with suggested actions and optimal path.

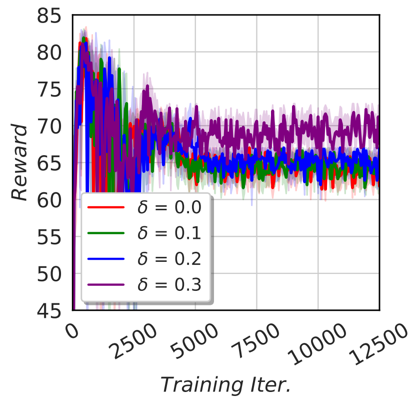
Determined Teacher: In this part, a teacher has a preferred action in every state with a probability of around 98%. As we observe in Figure 5.1a, these suggestions might help the student in reaching the target, but they are inefficient. For instance, if the student follows a square-wave sequence of actions as illustrated by the red line, she will be able to reach the target while following all of the teacher's suggestions exactly. Our objective is to allow the student to deviate from the teacher for a few steps, to find shorter routes.

By using PDPG, the student is able to find policies that are a mixture of the horizontal path and the square-wave route. For example, in Figure 5.1c, we have illustrated an instance of the student's optimized path with $\delta = 0.2$. The extent to which either policy is followed depends on values of δ and the KL-clipping parameter ρ. Figure 5.2a illustrates the student's total reward for different δ quantities without KL-clipping. We observe that as we increase δ, we allow the student to act more freely, hence she gets a higher reward. However, after 5000 training iterations, the reward remains at the same level with too much oscillation. Recalling the discussions of Section 5.4, this behavior indicates convergence to a stochastic policy.

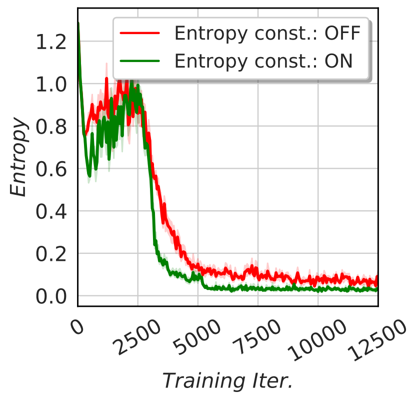
To reduce the oscillating behavior, we proposed adding an entropy constraint and KL-clipping. Figure 5.2b shows how adding the entropy constraint results in a more deterministic (i.e., lower entropy) policy. Also, in Figure 5.2c, we have added KL-clipping. As we decrease ρ, the student can totally disagree with the teacher in a larger proportion of the visited states, so she can find better policies with higher rewards. For different values of ρ, we see that the policy can converge to either a stochastic or deterministic one. For $\rho = 70, 75$, it converges to a deterministic horizontal line policy. With $\rho = 80, 85$, it learns to deterministically follow one $\sqcup$-shaped path followed by a horizontal route, and for $\rho = 95$, it follows a $\sqcup\sqcap\sqcup$-shaped path with a horizontal line at the end. Notice that even with $\rho = 100$, which means no clipping, the student is not exactly following the teacher. We also observe that for $\rho = 90$, it fails to converge to a deterministic policy. One justification for such a failure is that the student's policy is far better than the less-rewarding deterministic one, but not good enough to get to the next level of performance. Finally, Figure 5.2d shows how the λ and ζ values converge to their optimal values.

Less Confident Teacher: This experiment is designed to illustrate how a less confident teacher can still teach the student to follow some of his suggestions, but it will yield a lower level of confidence of the student. Figure 5.1b shows the suggested actions of the teacher; he is deterministic only in a subset of the states. For the rest, he does not have any information, so he suggests actions uniformly at random. The less confident teacher still has the square wave as the general idea (which is bad, just like the determined teacher), but also has extra randomness that points the student in even worse directions. In other words, the less confident teacher has a worse policy overall than the determined one. Recommending random actions causes the student to have more volatile behavior. We can observe this fact by comparing Figure 5.3a with 5.2a, where the student's converged policy produces a wider range of rewards for the less-confident teacher's case. Also, the average reward for this case is slightly lower, which can be explained by the inadequate information that the less-confident teacher provides for solving the task.

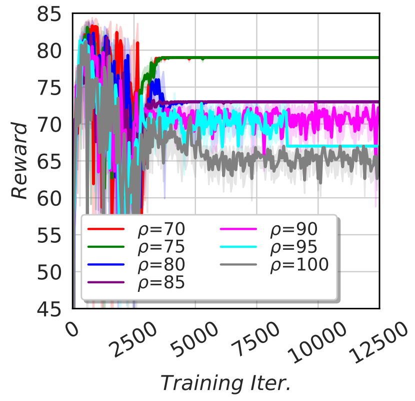
Figure 5.3b shows that adding KL-clipping helps in reducing the volatility, but one needs to choose a much smaller value for ρ (compare it with Figure 5.2c). Yet, even a small ρ does not necessarily result in a deterministic policy; for ρ as small as 0.4, the student has converged to a stochastic one.



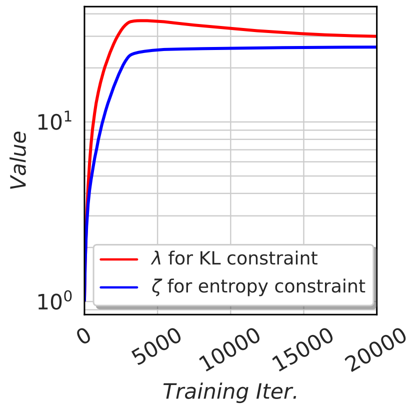
(a) The effect of δ on reward; no KL-clipping.



(b) The effect of entropy constraint; $\delta = 0.2$.

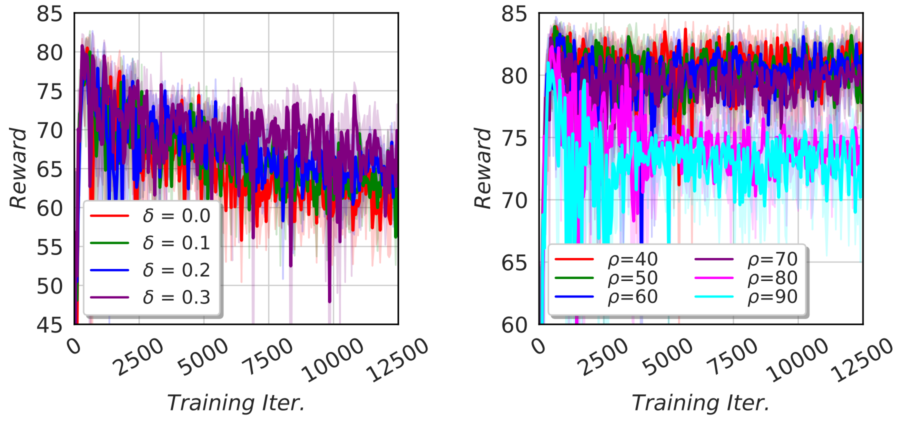


(c) Total reward for different ρ and $\delta = 0.2$.



(d) Convergence of λ and ζ; $\delta = 0.2$

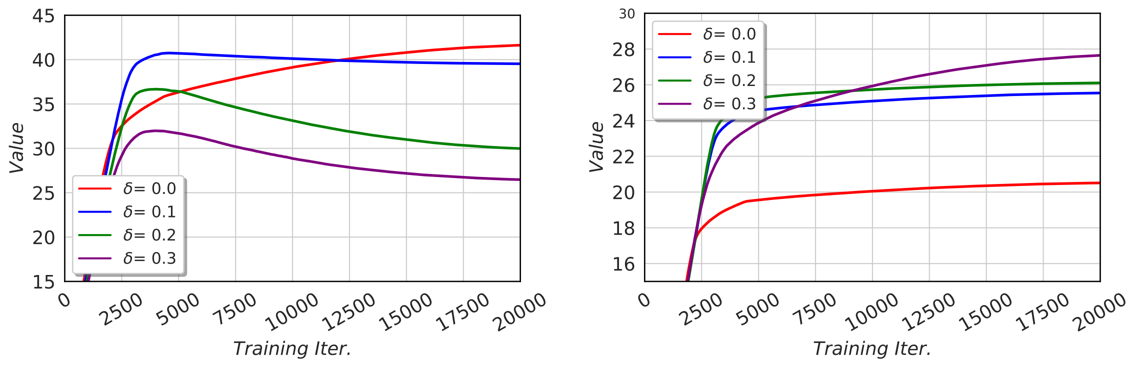
Figure 5.2 Performance of a student learning from the deterministic teacher.



(a) The effect of δ on reward; no KL-clipping. **(b)** Total reward for different ρ and $\delta = 0.2$

Figure 5.3 Performance of a student learning from the less confident teacher.

Effect of δ on Lagrange Multipliers In this part, we illustrate the convergence of both Lagrange multipliers λ and ζ to some steady values. From duality theory, we know that the converged values are a function of δ, and in Figure 5.4, we delineate these quantities for four different δ. In Figure 5.4a, we observe that as we increase λ (i.e., relax the constraint), we will converge to a smaller λ^*. A similar monotonic relation is observed in Figure 5.4b. The latter observation hypothesizes that in the square-wave experiment, larger δ values would bring more stochasticity to the optimal policy.

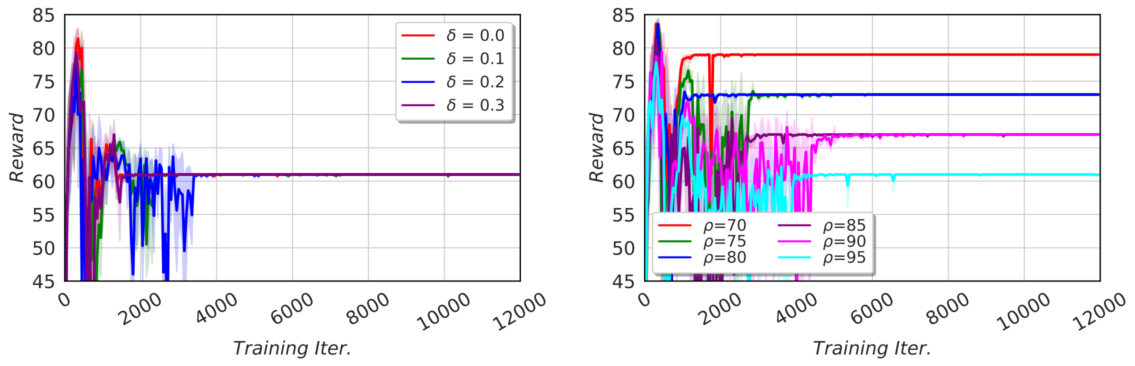


(a) λ convergence **(b)** ζ convergence

Figure 5.4 The effect of δ on Lagrange multipliers.

Effect of Using Reverse KL-divergence Note that throughout the experiments up to this point, we have used the forward KL-divergence. In this experiment, we intend to use the reverse KL constraint instead of the forward one to see how it affects learning. As we observe in Figure 5.5a, the student always converges to the teacher, no matter what the value of δ is. This is consistent with our theory that the student will converge to a sub-policy of the teacher. In fact, since the square-wave path is the only way that the teacher can reach the target, it will also be the optimal path for the student as well. Figure 5.5b shows that adding KL-clipping leads to different performance levels.

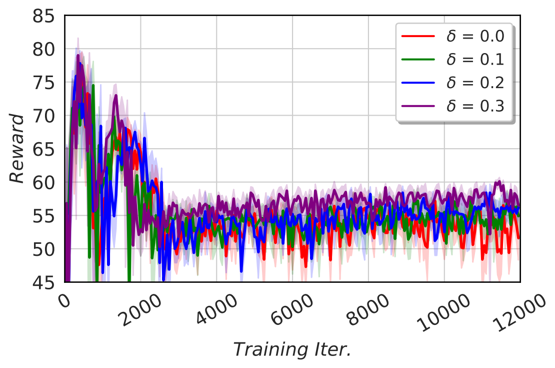
In Figure 5.6, we study the effect of using the reverse KL-constraint in learning from the less confident teacher. As we observe from Figure 5.6a, the student has converged to a policy similar to the square-wave policy, but with a few random actions. As we increase δ, we allow the student to disagree with the teacher more in the less confident states, so her policy becomes closer to the square-wave path (with reward 60). Also, Figure 5.6b shows a similar result as before on how KL-clipping may increase the reward.



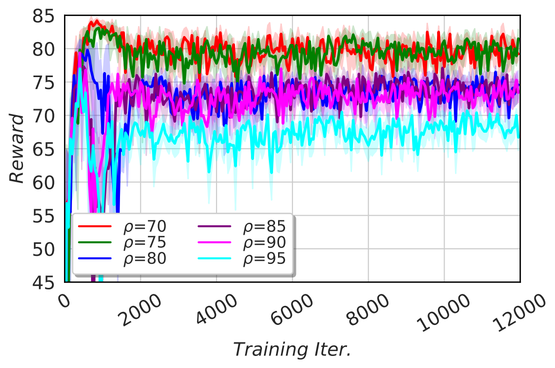
(a) The effect of δ on reward; no KL-clipping **(b)** Total reward for different ρ and $\delta = 0.2$

Figure 5.5 Performance of a student learning from the determined teacher using the reverse KL constraint.

Hellinger Constraint As we discussed in Section 5.4, one way to reduce the stochasticity of the converged policy might be using a finite distance measure such as the Hellinger metric in our constraints. Figure 5.7a shows the reward attained for the cases with and without the entropy constraint. As we see, the student has converged to a policy with reward 85, which corresponds to the horizontal path. Hence, she was not successful in learning from the teacher. We tried different configurations as well, but all exhibit similar behavior. Figure 5.7b shows that without using the entropy constraint, the student has relatively high



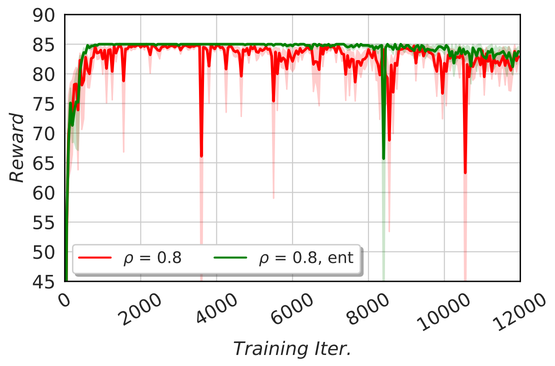
(a) The effect of δ on reward; no KL-clipping



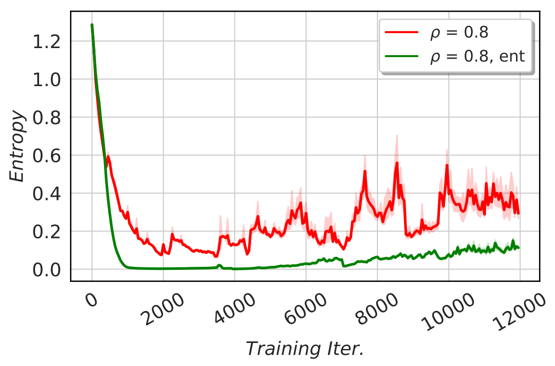
(b) Total reward for different ρ and $\delta = 0.2$

Figure 5.6 Performance of a student learning from the less confident teacher using the reverse KL constraint.

entropy, but she is still unable to follow the teacher.



(a) Reward gained with Hellinger constraint for $\delta = 0.2$.



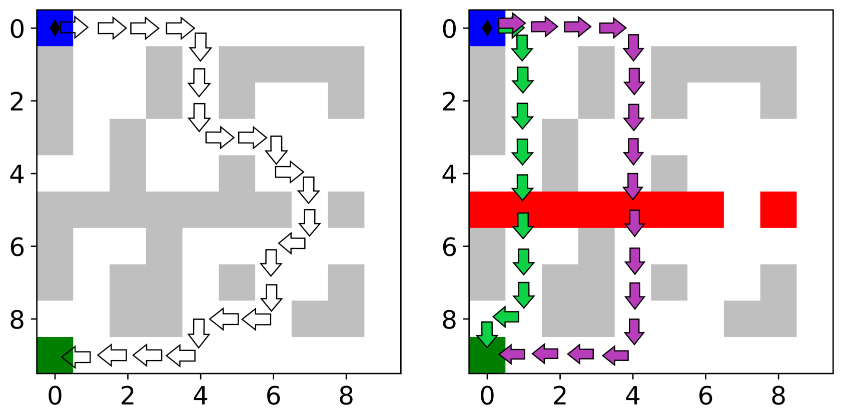
(b) Total reward for different ρ and $\delta = 0.2$

Figure 5.7 Performance of a student learning from determined teacher using the Hellinger constraint.

5.5.2 Wall Leaping

The purpose of this experiment is to show that PDPG can act as an improvement method when the student encounters a slightly modified environment. The teacher's reward structure is similar to the structure in the previous experiment, i.e., -1 for every step and $+100$ for reaching the target. However, the student comprehends that she can leap over some of the walls with a reward of -2. We use a vanilla policy gradient algorithm to train the teacher, which provides paths like the one illustrated in Figure 5.8a. If we allow the student to learn without any constraint, it will find the green path in Figure 5.8 with a

KL-divergence of ≈ 0.89. However, this is not what we are looking for since it is extremely different from the teacher. Instead, we use the PDPG algorithm to constrain the policy deviation with $\delta = 0.3$. Using this parameter, the student learns to follow the purple path, with a KL-divergence of ≈ 0.23.



(a) Teacher's environment. The optimal path found by the RL is demonstrated

(b) Student's environment. She can leap the red walls at a penalty. The paths found by RL (in green) and by PDPG (in purple) are illustrated

Figure 5.8 Wall-leaping teacher and student environments as well as their policies.

5.6 Related Work

Learning from a teacher is a well-studied problem in the literature on supervised learning [32] and imitation learning [75, 91]. However, we are not aware of any work using a teacher to control specific behaviors of a student. The typical use case of a student–teacher framework in RL is in "policy compression," where the objective is to train a student from a collection of well-trained RL policies. Policy distillation [73] and actor–mimic [68] are two methods that distill the trained RL agents, in a supervised learning fashion, into a unified policy of the student. In contrast, we follow a completely distinct objective, where a student is continually interacting with an environment and it only uses the teacher's signals as a guideline for shaping her policy.

Closest to ours, Schmitt et al. [76] propose "kickstarting RL," a method that uses the teacher's information for better training. Incorporating the idea of population-based training, they design a hand-crafted decreasing schedule of Lagrange multipliers, $\{\lambda^k\} \to 0$.

Nevertheless, the justification for such a schedule is not clearly visible. However, noticing that their problem is a special case of ours with $\delta = \infty$, our findings confirm the credibility of their approach, i.e., our findings indicate that $\lambda^* = \lim_{k\to\infty} \lambda^k_{min} = 0$ according to strong duality. This observation also conforms with the experimental findings of [76], and our theoretical results indicate that when there is no obligation on being similar to the teacher, the student is better off eventually operating independently. Similarly, their method only uses the teacher for faster learning.

Imposing certain constraints on the behavior of a policy is also a common problem in the context of "safe RL" [2, 22, 51]. Typically, these problems look for policies that avoid hazardous states either during training or execution. Our problem is different in that we follow another type of constraint, yet similar methods might be applied. Using a domain-specific programming language instead of neural networks can be an alternative method to add interpretability [95], but it lacks the numerous advantages inherent in end-to-end and differentiable learning. In an alternative direction, it is also possible to manipulate the policy shape by introducing auxiliary tasks or reward shaping [41]. Despite the simplicity of the latter approach, it has a very limited capability. For example, it is unclear how reward shaping can suggest directions similar to our square-wave teacher. In summary, we believe that our end-to-end method, by implicitly adding interpretable components, can partially alleviate the concerns related to the RL policies.

5.7 Concluding Remarks

In this paper, we introduce a new paradigm called corrective RL, which allows a "student" agent to learn to optimize its own policy while also staying sufficiently close to the policy of a "teacher." Our approach is motivated by the fact that practitioners may be reluctant to adopt the policies proposed by RL algorithms if they differ too much from the status quo. Even if the RL policy produces an impressive expected return, this may not be satisfactory evidence to switch the operation of a billion-dollar company to a policy found by an RL. We believe that corrective RL provides a straightforward remedy by constraining how far the new policy can deviate from the old one or another desired, target policy. Doing so will help reduce the stresses of adopting a novel policy.

We believe that, with further extensions, corrective RL has the potential to address some of RL's interpretability challenges. Using more advanced optimization algorithms, studying different distance measures, considering continuous-action problems, and having multiple teachers represent fruitful avenues for future research.

Appendix

5.A Convergence of PDPG Algorithm

In this Appendix, we first derive the gradients necessary for the PDPG algorithm. Subsequently, we provide convergence proofs for PDPG Algorithm.

5.A.1 Computing the Gradients

The Lagrangian function in the optimization problem (5.4) can be re-written as

$$\begin{aligned} L(\theta, \lambda) &= \sum_{\tau \in \mathcal{T}} \mathbb{P}_\theta(\tau) J(\tau) + \lambda \sum_{\tau \in \mathcal{T}} \mathbb{P}_\theta(\tau) \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)} - \lambda \delta \\ &= \sum_{\tau \in \mathcal{T}} \mathbb{P}_\theta(\tau) \left(J(\tau) + \lambda \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)} \right) - \lambda \delta. \end{aligned} \tag{5.15}$$

Recall that $\mathcal{T}$ is the set of all trajectories under all admissible policies. By taking the gradient of $L(\theta, \lambda)$ with respect to θ, we have:

$$\begin{aligned} \nabla_\theta L(\theta, \lambda) &= \sum_{\tau \in \mathcal{T}} \nabla_\theta \mathbb{P}_\theta(\tau) \left(J(\tau) + \lambda \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)} \right) + \mathbb{P}_\theta(\tau) \left(\lambda \nabla_\theta \log \mathbb{P}_\theta(\tau) \right) \\ &= \sum_{\tau \in \mathcal{T}} \mathbb{P}_\theta(\tau) \nabla_\theta \log \mathbb{P}_\theta(\tau) \left(J(\tau) + \lambda \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)} + \lambda \right) \\ &= \mathbb{E}_{\mathcal{T}} \left[\nabla_\theta \log \mathbb{P}_\theta(\tau) \left(J(\tau) + \lambda \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)} + \lambda \right) \right], \end{aligned} \tag{5.16}$$

and the term $\nabla_\theta \log \mathbb{P}_\theta(\tau)$ can be simplified as

$$\begin{aligned}
\nabla_\theta \log \mathbb{P}_\theta(\tau) &= \nabla_\theta \left(\log P_0(x_0) + \sum_{t=0}^{H-1} \log P(x_{t+1}|x_t, a_t) + \sum_{t=0}^{H-1} \log \pi_S(a_t|x_t;\theta) \right) \\
&= \sum_{t=0}^{H-1} \nabla_\theta \log \pi_S(a_t|x_t;\theta) \\
&= \sum_{t=0}^{H-1} \frac{\nabla_\theta \pi_S(a_t|x_t;\theta)}{\pi_S(a_t|x_t;\theta)}. \qquad (5.17)
\end{aligned}$$

The gradient of $L(\theta, \lambda)$ with respect to λ is

$$\nabla_\lambda L(\theta,\lambda) = \sum_{\tau \in \mathcal{T}} \mathbb{P}_\theta(\tau) \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)} - \delta = D_{\mathrm{KL}}(\mathbb{P}_\theta(\tau) \parallel \mathbb{P}_\phi(\tau)) - \delta. \qquad (5.18)$$

By using a set of sample trajectories $\{\tau_j, j = 1, \ldots, N\}$ generated under the student policy, one can approximate the gradients (5.16) and (5.18) as

$$\begin{aligned}
\nabla_\theta L(\theta,\lambda) &\approx \frac{1}{N} \sum_{j=1}^{N} \left[\nabla_\theta \log \mathbb{P}_\theta(\tau_j) \left(J(\tau_j) + \lambda \log \frac{\mathbb{P}_\theta(\tau_j)}{\mathbb{P}_\phi(\tau_j)} + \lambda \right) \right], \\
\nabla_\lambda L(\theta,\lambda) &\approx \hat{D}_{\mathrm{KL}}(\theta \parallel \phi) - \delta = \frac{1}{N} \sum_{j=1}^{N} \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)} - \delta,
\end{aligned}$$

which are the update rules that will be used later on, in (5.7) and (5.8).

5.A.2 Convergence Analysis of PDPG for (OPT-R)

Before starting the proof of Theorem 5.3.3, noting the definition of $\nabla_\theta L(\theta,\lambda)$ and $\nabla_\lambda L(\theta,\lambda)$, one can make the following observations:

Lemma 5.A.1. *Under Assumption 5.3.1, the following holds:*

i) $\nabla_\theta \log \mathbb{P}_\theta(\tau)$ is Lipschitz continuous in θ, which further implies that

$$\|\nabla_\theta \log \mathbb{P}_\theta(\tau)\|^2 \le \kappa_1(\tau) \left(1 + \|\theta\|^2\right) \qquad (5.19)$$

for some $\kappa_1(\tau) < \infty$.

ii) $\nabla_\theta L(\theta, \lambda)$ *is Lipschitz continuous in* θ*, which further implies that*

$$\|\nabla_\theta L(\theta, \lambda)\|^2 \leq \kappa_2 \left(1 + \|\theta\|^2\right) \tag{5.20}$$

for some constant $\kappa_2 < \infty$*.*

iii) $\nabla_\lambda L(\theta, \lambda)$ *is Lipschitz continuous in* λ*.*

Proof. Recall from (5.17) that $\nabla_\theta \log \mathbb{P}_\theta(\tau) = \sum_{t=0}^{H-1} \nabla_\theta \pi_S(a_t|x_t;\theta)/\pi_S(a_t|x_t;\theta)$ whenever we have
$\pi_S(a_t|x_t;\theta) > \psi$ for all t and for some $\psi > 0$. Assumption 5.3.1 indicates that $\nabla_\theta \pi_S(a_t|x_t;\theta)$ is $\mathscr{L}$-Lispchitz continuous in θ. Then using the fact the sum of the product of (bounded) Lipschitz functions is Lipschitz itself, one can conclude the Lipschitz continuity of $\nabla_\theta \log \mathbb{P}_\theta(\tau)$, and we denote by L_1 its finite Lipschitz constant. Also, noting that $H < \infty$ w.p. 1, then $\nabla_\theta \log \mathbb{P}_\theta(\tau) < \infty$ w.p. 1. The Lipschitz continuity implies that for any fixed $\theta_0 \in \Theta$,

$$\|\nabla_\theta \log \mathbb{P}_\theta(\tau)\| \leq \|\nabla_\theta \log \mathbb{P}_\theta(\tau)|_{\theta=\theta_0}\| + L_1\|\theta - \theta_0\| \leq K_1(\tau)(1 + \|\theta\|). \tag{5.21}$$

The first inequality follows from the linear growth condition of Lipschitz functions and the last one holds for a suitable value of $K_1(\tau) := \max\{L_1, \|\nabla_\theta \log \mathbb{P}_\theta(\tau)|_{\theta=\theta_0}\| + L_1\|\theta_0\|\} < \infty$. Taking the square of both sides of (5.21) yields (5.19) with $\kappa_1(\tau) := 2(K_1(\tau))^2 < \infty$.

Since $\mathbb{P}_\theta(\tau)$ and $\log \mathbb{P}_\theta(\tau)$ are continuously differentiable in θ whenever $\mathbb{P}_\theta(\tau) > 0$, the Lipschitz continuity of $\nabla_\theta L(\theta, \lambda)$ can be investigated, from its definition (5.16), as the sums of products of (bounded) Lipschitz functions. From the definition (5.16), and recalling Assumption 5.2.1 and the compactness of Θ, one can verify the validity of (5.20) with

$$\kappa_2 = \mathbb{E}_\tau \left[\kappa_1(\tau) \left(\frac{C_{max}}{1-\gamma} + \lambda_{max} \max_{\theta \in \Theta} \log \frac{\mathbb{P}_\theta(\tau)}{\mathbb{P}_\phi(\tau)}\right)\right] < \infty. \tag{5.22}$$

Finally, *iii)* immediately follows from the fact that $\nabla_\lambda L(\theta, \lambda)$ is a constant function of λ. □

We use the standard procedure for proving the convergence of the PDPG algorithm. The proof steps are common for stochastic approximation methods and we refer the reader to [11, 21] and references therein for more details. We summarize the scheme of the proof in the following steps:

1. **Tracking o.d.e.**: Under Assumption 5.3.2, one can view the PDPG as a two-time-scale stochastic approximation method. Then, using the results of Section 6 of [12], we show that the sequence of (θ^k, λ^k) converges almost surely to a stationary point (θ^*, λ^*) of the corresponding continuous-time dynamical system.

2. **Lyapunov Stability**: By using Lyapunov analysis, we show that the continuous-time system is locally asymptotically stable at a first-order stationary point.

3. **Saddle Point Analysis**: Since we have used the Lagrangian as the Lyapunov function, it implies the system is stable in the stationary point of the Lagrangian, which is, in fact, a local saddle point. Finally, we show that with an appropriate initial policy, the policy converges to a local optimal solution θ^* for the OPT-R.

First, let us denote by $\Psi_\Xi\left[f(\xi)\right]$ the right directional derivative of $\Gamma_\Xi(\xi)$ in the direction of $f(\xi)$, defined as

$$\Psi_\Xi\left[f(\xi)\right] := \lim_{\alpha \downarrow 0} \frac{\Gamma_\Xi\left[\xi + \alpha f(\xi)\right] - \Gamma_\Xi\left[\xi\right]}{\alpha}$$

for any compact set Ξ and $\xi \in \Xi$.

Since θ converges on a faster time-scale than λ by Assumption 5.3.2, one can write the θ-update rule (5.7) with a relation that is invariant to λ:

$$\theta^{k+1} = \Gamma_\Theta\left[\theta^k - \alpha_1(k)\left(\frac{1}{N}\sum_{j=1}^{N} \nabla_\theta \log \mathbb{P}_\theta(\tau_j^k)\bigg|_{\theta=\theta^k}\left(J(\tau_j^k) + \lambda \log\frac{\mathbb{P}_\theta(\tau_j^k)}{\mathbb{P}_\phi(\tau_j^k)} + \lambda\right)\right)\right].$$

Consider the continuous-time dynamics of $\theta \in \Theta$ defined as

$$\dot{\theta} = \Psi_\Theta\left[-\nabla_\theta L(\theta, \lambda)\right], \tag{5.23}$$

where by using the right directional derivative $\Psi_\Theta\left[-\nabla_\theta L(\theta, \lambda)\right]$ in the gradient descent algorithm for θ, the gradient will point in the descent direction of $L(\theta, \lambda)$ along the boundary of Θ (denoted by $\partial\Theta$) whenever the θ-update hits the boundary. We refer the interested reader to Section 5.4 of [12] for discussions about the existence of the limit in (5.23).

Since λ converges in the slowest time-scale, the λ-update rule (5.8) can be re-written for a converged value $\theta^*(\lambda)$ as

$$\lambda^{k+1} = \Gamma_\Lambda \left[\lambda^k + \alpha_2(k) \left(\frac{1}{N} \sum_{j=1}^{N} \log \frac{\mathbb{P}_{\theta^*(\lambda)}(\tau_j^k)}{\mathbb{P}_\phi(\tau_j^k)} - \delta \right) \right].$$

Consider the continuous-time dynamics corresponding to λ, i.e.

$$\dot{\lambda} = \Psi_\Lambda \left[\nabla_\lambda L(\theta, \lambda) \right], \tag{5.24}$$

where by using $\Psi_\Lambda \left[\nabla_\lambda L(\theta, \lambda) \right]$ in the gradient ascent algorithm, the gradient will point in the ascent direction along the boundary of Λ (denoted by $\partial\Lambda$) whenever the λ-update hits the boundary.

We prove Theorem 5.3.3 next.

Proof. **Convergence of the θ-update**: First, we need to show that the assumptions of Lemma 1 in Chapter 6 of [12] hold for the θ-update and an arbitrary value of λ. Let us justify these assumptions: (i) the Lipschitz continuity follows from Lemma 5.A.1, and (ii) the step-size rules follow from Assumption 5.3.2. (iii) For an arbitrary value λ, one can write the θ-update as a stochastic approximation, i.e.,

$$\theta^{k+1} = \Gamma_\Theta \left[\theta^k + \alpha_1(k) \left(-\nabla_\theta L(\theta, \lambda)|_{\theta=\theta^k} + M_{\theta_{k+1}} \right) \right], \tag{5.25}$$

where

$$M_{\theta^{k+1}} = \nabla_\theta L(\theta, \lambda)|_{\theta=\theta^k} - \frac{1}{N} \sum_{j=1}^{N} \nabla_\theta \log \mathbb{P}_\theta(\tau_j^k) \Big|_{\theta=\theta^k} \left(J(\tau_j^k) + \lambda^k \log \frac{\mathbb{P}_\theta(\tau_j^k)}{\mathbb{P}_\phi(\tau_j^k)} + \lambda^k \right). \tag{5.26}$$

For $M_{\theta^{k+1}}$ to be a Martingale difference error term, we need to show that its expectation with respect to the filtration $\mathcal{F}_\theta^k = \sigma(\theta^m, M_{\theta^m}, m \le k)$ is zero and that it is square integrable with $\mathbb{E}\left[\|M_{\theta^{k+1}}\|^2 | \mathcal{F}_\theta^k \right] \le \kappa^k (1 + \|\theta^k\|^2)$ for some κ^k. Since the trajectories $\mathcal{T}^k$ are generated from the probability mass function $\mathbb{P}_{\theta^k}(\cdot)$, it immediately follows that $\mathbb{E}\left[M_{\theta^{k+1}}|\mathcal{F}_\theta^k\right] = 0$. Also, we have:

$$\begin{aligned}
&\|M_{\theta^{k+1}}\|^2\\
&\leq 2\|\nabla_\theta L(\theta,\lambda)|_{\theta=\theta^k}\|^2 + \frac{2}{N^2}\left(\frac{C_{max}}{1-\gamma} + \lambda_{max}\left(D_{max}^k+1\right)\right)^2 \left\|\sum_{j=1}^N \nabla_\theta \log \mathbb{P}_\theta(\tau_j^k)\big|_{\theta=\theta^k}\right\|^2\\
&\leq 2\kappa_2^k\left(1+\|\theta^k\|^2\right) + \frac{2^N}{N^2}\left(\frac{C_{max}}{1-\gamma} + \lambda_{max}\left(D_{max}^k+1\right)\right)^2\left(\sum_{j=1}^N \kappa_1^k(\tau_j^k)\left(1+\|\theta^k\|^2\right)\right)\\
&\leq \kappa^k\left(1+\|\theta^k\|^2\right),
\end{aligned}$$

where

$$\begin{aligned}
D_{max}^k &= \max_{1\leq j\leq N} \log\frac{\mathbb{P}_\theta(\tau_j^k)}{\mathbb{P}_\phi(\tau_j^k)}, \quad \text{and}\\
\kappa^k &= 2\kappa_2^k + \frac{2^N}{N}\max_{1\leq j\leq N}\kappa_1^k(\tau_j^k)\left(\frac{C_{max}}{1-\gamma} + \lambda_{max}\left(D_{max}^k+1\right)\right)^2 < \infty.
\end{aligned}$$

The first and second inequality uses the relation $\|\sum_{i=1}^N a_i\|^2 \leq 2^{N-1}(\sum_{i=1}^N \|a\|^2)$. Also, the second one uses the results of Lemma 5.A.1. Finally, the boundedness of κ^k follows from Assumption 5.2.1 and having $\kappa_1^k < \infty$, $\kappa_2^k(\tau_j^k) < \infty$ w.p. 1. Finally, (iv) $\sup_k \|\theta^k\| < \infty$ almost surely, because all θ^k are within the compact set Θ. Hence, by Theorem 2 of Chapter 2 in [12], the sequence $\{\theta^k\}$ converges almost surely to a (possibly sample path dependent) internally chain transitive invariant set of o.d.e. (5.23).

For a given λ, define the Lyapunov function

$$\mathcal{L}_\lambda(\theta) = L(\theta,\lambda) - L(\theta^*,\lambda), \tag{5.27}$$

where $\theta^* \in \Theta$ is a local minimum point. For the sake of simplifying the proof, let us consider that θ^* is an isolated local minimum point, i.e., there exists r such that for all $\theta \in \mathbb{B}_r(\theta^*)$, $\mathcal{L}_\lambda(\theta) > \mathcal{L}_\lambda(\theta^*)$. This means that the Lyapunov function $\mathcal{L}_\lambda(\theta)$ is locally positive definite, i.e., $\mathcal{L}_\lambda(\theta^*) = 0$ and $\mathcal{L}_\lambda(\theta) > 0$ for $\mathbb{B}_r \setminus \{\theta^*\}$.

If we establish the negative semi-definiteness of $d\mathcal{L}_\lambda(\theta)/dt \leq 0$, then we can use the

Lyapunov stability theorems to show the convergence of the dynamical system. Consider the time derivative of the corresponding continuous-time system for θ, i.e.,

$$\frac{d\mathcal{L}_\lambda(\theta)}{dt} = \frac{dL(\theta,\lambda)}{dt} = (\nabla_\theta L(\theta,\lambda))^T \Psi_\Theta(-\nabla_\theta L(\theta,\lambda)). \tag{5.28}$$

Consider two cases:

i) For a fixed $\theta_0 \in \Theta$, there exists $\alpha_0 > 0$ such that the update $\theta_0 - \alpha\nabla_\theta L(\theta,\lambda)|_{\theta=\theta_0} \in \Theta$ for all $\alpha \in (0,\alpha_0]$. In this case, $\Psi_\Theta(-\nabla_\theta L(\theta,\lambda)) = -\nabla_\theta L(\theta,\lambda)$, which further implies that

$$\frac{dL(\theta_0,\lambda)}{dt} = -\|\nabla_\theta L(\theta,\lambda)|_{\theta=\theta_0}\|^2 \leq 0,$$

and this quantity is non-zero as long as $\|\Psi_\Theta(-\nabla_\theta L(\theta,\lambda))\| \neq 0$.

ii) For fixed $\theta_0 \in \Theta$ and any $\alpha_0 > 0$, there exists $\alpha \in (0,\alpha_0]$ such that $\theta_\alpha := \theta_0 - \alpha\nabla_\theta L(\theta,\lambda)|_{\theta=\theta_0} \notin \Theta$. The projection $\Gamma_\Theta(\theta_\alpha) = \arg\min_{\theta\in\Theta} \frac{1}{2}\|\theta - \theta_\alpha\|^2$ maps θ_α to a point in $\partial\Theta$. This projection is single-valued because of the compactness and convexity of Θ, and we denote the projected point by $\bar{\theta}_\alpha \in \Theta$. Consider $\alpha \downarrow 0$, then

$$\begin{aligned}(\nabla_\theta L(\theta,\lambda))^T \Psi_\Theta(-\nabla_\theta L(\theta,\lambda)) &= \lim_{\alpha\downarrow 0} \frac{(\theta-\theta_\alpha)^T(\bar{\theta}_\alpha - \theta)}{\eta} \\ &= \lim_{\alpha\downarrow 0} \frac{-\|\bar{\theta}_\alpha - \theta|^2}{\eta^2} + \frac{(\bar{\theta}_\alpha - \theta_\alpha)^T(\bar{\theta}_\alpha - \theta)}{\eta^2} \leq 0,\end{aligned}$$

where the last inequality follows from the Projection Theorem (see Proposition 1.1.9 of [10]). Again, one can verify that the time-derivative quantity is non-zero as long as $\|\Psi_\Theta(-\nabla_\theta L(\theta,\lambda))\| \neq 0$.

In summary, $d\mathcal{L}_\lambda(\theta)/dt \leq 0$ and this quantity is nonzero as long as $\|\Psi_\Theta(-\nabla_\theta L(\theta,\lambda))\| \neq 0$. Then by *LaSalle's Local Invariant Set Theorem* (see, e.g., Theorem 3.4 of [80]), we conclude that the dynamical system tends to the largest positive invariant set within $\mathbb{M}_\theta := \{\theta : \|\Psi_\Theta(-\nabla_\theta L(\theta,\lambda))\| = 0\}$. Notice that $\theta^* \in \mathbb{M}_\theta$. Let $l > 0$ be equal to

$$\min\{\mathcal{L}_\lambda(\theta) : \|\Psi_\Theta(-\nabla_\theta L(\theta,\lambda))\| = 0, \theta \in \mathbb{B}_r(\theta^*) \setminus \theta^*\}.$$

Then every trajectory starting from the attraction region $\{\theta \in \mathbb{B}_r(\theta^*) | \mathcal{L}_\lambda(\theta) < l\}$ will tend to the local minimum θ^*. Since we chose θ^* to be arbitrary, this holds for all local minima. Hence, using Corollary 4 of Chapter 2 in [12], we conclude that if the initial policy θ^0 is within the attraction region of a local minimum point θ^*, then it will converge to it almost surely.

Remark 5.A.2. *The case in which θ^* is not isolated can be handled similarly, with the minor difference that the convergence happens to a set of optimal points instead of to a single point.*

Convergence of the λ-update: We need to show that the assumptions of Theorem 2 in Chapter 6 of [12] hold for the two-time-scale stochastic approximation theory. Let us verify the validity of these assumptions: (i) $\nabla_\lambda L(\theta, \lambda)$ is a Lipschitz function in λ from Lemma 5.A.1, and (ii) step-size rules follow from Assumption 5.3.2. (iii) Since λ converges in a slower time-scale, we have $\|\theta^{k,i} - \theta^*(\lambda^k)\| \to 0$ almost surely as $i \to \infty$, which, according to the Lipschitz continuity of $\nabla_\lambda L(\theta, \lambda)$, implies that

$$\|\nabla_\lambda L(\theta,\lambda)|_{\theta=\theta^{k,i},\lambda=\lambda^k} - \nabla_\lambda L(\theta,\lambda)|_{\theta=\theta^*(\lambda^k),\lambda=\lambda^k}\| \to 0 \quad \text{as } i \to \infty. \tag{5.29}$$

Hence the λ-update can be written as

$$\lambda^{k+1} = \Gamma_\Lambda\Big[\lambda^k + \alpha_2(k)\left(\nabla_\lambda L(\theta,\lambda)|_{\theta=\theta^*(\lambda^k),\lambda=\lambda^k} + M_{\lambda^{k+1}}\right)\Big],$$

where

$$M_{\lambda^{k+1}} = -\nabla_\lambda L(\theta,\lambda)|_{\theta=\theta^*(\lambda^k),\lambda=\lambda^k} + \Big(\frac{1}{N}\sum_{j=1}^{N} \log\frac{\mathbb{P}_{\theta^*(\lambda^k)}(\tau_j^k)}{\mathbb{P}_\phi(\tau_j^k)} - \delta\Big). \tag{5.30}$$

From (5.30), we can verify that $\mathbb{E}\left[M_{\lambda^{k+1}}|\mathcal{F}_\lambda^k\right] = 0$, where $\mathcal{F}_\lambda^k = \sigma(\lambda^m, M_{\lambda^m}, m \le k)$ is a filtration of λ generated by different independent trajectories. Also, we have:

$$\|M_{\lambda^{k+1}}\|^2 \le 2\|\nabla_\lambda L(\theta,\lambda)|_{\lambda=\lambda^k}\|^2 + \frac{2^N}{N}\left(\max_{1\le j\le N}\left|\log\frac{\mathbb{P}_{\theta^*(\lambda^k)}(\tau_j^k)}{\mathbb{P}_\theta(\tau_j^k)} - \delta\right|\right)^2 < \infty.$$

Hence, $M_{\lambda^{k+1}}$ is a Martingale difference error. Also, (v) $\sup\{\lambda^k\} < \infty$. Recall that from the convergence analysis of the θ-update for a λ^k, we know that $\theta^*(\lambda^k)$ is an asymptotically stable point. Then by Theorem 2 of Chapter 6 in [12], we can conclude that (θ^k, λ^k) converges almost surely to $(\theta^*(\lambda^*), \lambda^*)$, where λ^* belongs to an internally chain transitive invariant set of (5.24).

Define the Lyapunov function:

$$\mathcal{L}(\lambda) = -L(\theta^*(\lambda), \lambda) + L(\theta^*(\lambda^*), \lambda^*),$$

where λ^* is a local maximum point, i.e., there exists r such that for any $\lambda \in \mathcal{B}_r(\lambda^*)$, the Lyapunov function $\mathcal{L}(\lambda)$ is positive definite. We can follow similar lines of arguments as we did for the θ-update to show that $\frac{d\mathcal{L}(\lambda)}{dt} \leq 0$ and this quantity is non-zero as long as $\Psi_\Lambda(-\nabla_\lambda L(\theta^*(\lambda), \lambda)) \neq 0$. Then by using the results of LaSalle's Local Invariant Set Theorem, we can establish the convergence of the dynamical system to the largest invariant set within

$$\mathbb{M}_\lambda := \{\lambda : \Psi_\Lambda(-\nabla_\lambda L(\theta^*(\lambda), \lambda)) = 0\}.$$

This means that $\lambda^* \in \mathbb{M}_\lambda$ is a stationary point. Let

$$l = \min\{\mathcal{L}(\lambda) : \Psi_\Lambda(-\nabla_\lambda L(\theta^*(\lambda), \lambda)) = 0, \lambda \in \mathcal{B}_r(\lambda^*) \setminus \lambda^*\}.$$

Then, every trajectory starting with λ^0 in $\{\lambda \in \mathcal{B}_r(\lambda^*) : \mathcal{L}(\lambda) < l\}$ will tend to λ^* w.p. 1.

Saddle Point Analysis: By denoting $\theta^* = \theta^*(\lambda^*)$, we want to show that (θ^*, λ^*) is, in fact, a saddle point of the Lagrangian $L(\theta, \lambda)$. Recall that, as we proved in the convergence the of θ-update, θ^* is a local minimum of $L(\theta, \lambda)$ within a sufficiently small ball around itself, i.e., there exists $r > 0$ such that

$$L(\theta^*, \lambda^*) \leq L(\theta, \lambda^*), \quad \forall \theta \in \Theta \cap \mathcal{B}_r(\theta^*). \tag{5.31}$$

It is easy to verify that θ^* is a feasible solution of (OPT-R) whenever $\lambda^* \in [0, \lambda_{max})$, i.e.

$$D_{KL}(\theta^* \parallel \phi) \leq \delta. \tag{5.32}$$

To show this, assume for a contradiction that $D_{KL}(\theta^* \parallel \phi) - \delta > 0$. Then,

$$\begin{aligned}
\Psi_\Lambda \left[\nabla_\lambda L(\theta, \lambda)|_{\theta=\theta^*, \lambda=\lambda^*}\right] &= \lim_{\alpha \downarrow 0} \frac{\Gamma_\Lambda \left[\lambda^* + \alpha \nabla_\lambda L(\theta, \lambda)|_{\theta=\theta^*, \lambda=\lambda^*}\right] - \Gamma_\Lambda \left[\lambda^*\right]}{\alpha} \\
&= \lim_{\alpha \downarrow 0} \frac{\Gamma_\Lambda \left[\lambda^* + \alpha \left(D_{KL}(\theta^* \parallel \phi) - \delta\right)\right] - \Gamma_\Lambda \left[\lambda^*\right]}{\alpha} \\
&= D_{KL}(\theta^* \parallel \phi) - \delta > 0,
\end{aligned}$$

which contradicts the fact that $\Psi_\Lambda \left[\nabla_\lambda L(\theta, \lambda)|_{\theta=\theta^*, \lambda=\lambda^*}\right] = 0$. Notice that the feasibility cannot be verified when $\lambda^* = \lambda_{max}$, because $\Psi_\Lambda \left[\nabla_\lambda L(\theta, \lambda)|_{\theta=\theta^*(\lambda_{max}), \lambda=\lambda_{max}}\right] = 0$ when $D_{KL}(\theta^* \parallel \phi) > \delta$. In this case, we increase λ_{max} (e.g., we set $\lambda_{max} \leftarrow 2\lambda_{max}$ in our algorithm) if such a behavior happens until it converges to an interior point of $[0, \lambda_{max}]$.

In addition, the complementary slackness condition

$$\lambda^*(D_{KL}(\theta^* \parallel \phi) - \delta) = 0 \tag{5.33}$$

holds. To show this, we only need to verify that $D_{KL}(\theta^* \parallel \phi) < \delta$ yields $\lambda^* = 0$. For a contradiction, suppose that $\lambda^* \in (0, \lambda_{max})$. Then, we have

$$\Psi_\Lambda \left[\nabla_\lambda L(\theta, \lambda)|_{\theta=\theta^*(\lambda^*), \lambda=\lambda^*}\right] = D_{KL}(\theta^* \parallel \phi) - \delta < 0,$$

which contradicts the fact that $\Psi_\Lambda \left[\nabla_\lambda L(\theta, \lambda)|_{\theta=\theta^*, \lambda=\lambda^*}\right] = 0$, meaning that $\lambda^* = 0$ in this case. Hence, we have:

$$\begin{aligned}
L(\theta^*, \lambda^*) &= V_{\theta^*}(x_0) + \lambda^* \left(D_{KL}(\theta^* \parallel \phi) - \delta\right) \\
&= V_{\theta^*}(x_0) \\
&\geq V_{\theta^*}(x_0) + \lambda \left(D_{KL}(\theta^* \parallel \phi) - \delta\right) = L(\theta^*, \lambda).
\end{aligned} \tag{5.34}$$

From (5.31) and (5.34), we observe that (θ^*, λ^*) is a saddle point of $L(\theta, \lambda)$, so according to the saddle point theorem, θ^* is a local minimum of (OPT-R). Recall that the result of Theorem 5.3.3 depends on the initial values for θ^0 and λ^0, so the convergence to a local minimum is sample path depenedant. $\square$

5.A.3 Proof of Corollary 5.3.4

Proof. From the convergence analysis of the θ-update, we know that $\{\theta^k\}$ converges almost surely to the largest invariant set within $\mathbb{M}_\theta$, and similarly, $\{\lambda^k\}$ converges almost surely to the largest invariant set within $\mathbb{M}_\lambda$. We also know from (5.32) that θ^* is a feasible point of (OPT-R). When $\lambda^* = 0$, then $L(\theta^*, \lambda^*) = V_{\theta^*}(x_0)$. Also, for $\lambda^* > 0$, the complementary slackness condition (5.33) implies $D_{KL}(\theta^* \parallel \phi) = \delta$. Hence $\nabla_\theta D_{KL}(\theta \parallel \phi)|_{\theta=\theta^*} = 0$, which in turn, means that

$$\nabla_\theta L(\theta, \lambda^*)|_{\theta=\theta^*} = \nabla_\theta V_\theta(x_0)|_{\theta=\theta^*} + \lambda^* \nabla_\theta D_{KL}(\theta \parallel \phi)|_{\theta=\theta^*} = \nabla_\theta V_\theta(x_0)|_{\theta=\theta^*}. \tag{5.35}$$

Hence, for a θ^* located in the interior of Θ, we have $\nabla_\theta L(\theta, \lambda^*)|_{\theta=\theta^*} = \nabla_\theta V_\theta(x_0)|_{\theta=\theta^*} = 0$, so it is a first-order stationary point of (OPT-R). However, if $\theta^* \in \partial\Theta$, it is possible to have $\|\nabla_\theta L(\theta, \lambda^*)|_{\theta=\theta^*}\| \neq 0$. $\square$

Remark 5.A.3. *In practice, we choose the projection set Θ large enough so that the latter case (convergence to boundary) will not happen. For example, assuring that the weights of a neural network do not diverge is a sufficient criterion to use instead of the projection operator Γ_Θ.*

5.A.4 Equivalent Results for (OPT-F)

A similar PDPG algorithm to the one proposed in Algorithm 8 can solve (OPT-F), only requiring a slight modification of rules (5.7) and (5.8) as

$$\theta^{k+1} = \Gamma_\Theta\Big[\theta^k - \alpha_1(k)\Big(\frac{1}{N}\sum_{j=1}^{N} \nabla_\theta \log \mathbb{P}_\theta(\tau_j^k)\big|_{\theta=\theta^k}\big(J(\tau_j^k) + \lambda^k\, IS(\tau_j^k) \log \frac{\mathbb{P}_\phi(\tau_j^k)}{\mathbb{P}_\theta(\tau_j^k)} - \lambda^k\big)\Big)\Big]$$

$$\lambda^{k+1} = \Gamma_\Lambda\Big[\lambda^k + \alpha_2(k)\Big(\frac{1}{N}\sum_{j=1}^{N} IS(\tau_j^k) \log \frac{\mathbb{P}_\phi(\tau_j^k)}{\mathbb{P}_\theta(\tau_j^k)} - \delta\Big)\Big],$$

where $IS(\tau_j^k) = \mathbb{P}_\phi(\tau_j^k)/\mathbb{P}_\theta(\tau_j^k)$ is the importance sampling weight added to account for the bias introduced by sampling under the student's policy. To ensure a well-defined (OPT-F), we need the following assumption:

Assumption 5.A.4. ***Well-defined*** (OPT-F)*: for any state–action pair $(x, a) \in \mathcal{X} \times \mathcal{A}$ with $\pi_S(x, a) = 0$, we have $\pi_T(x, a) = 0$.*

This assumption ensures a similar criterion to that of Assumption 5.2.1, but notice that in this case, the student might take any action, regardless of the teacher's policy. Exactly the same steps can be taken, virtually verbatim, to prove the following convergence property of the PDPG algorithm for (OPT-F).

Theorem 5.A.5. *Under Assumptions 5.3.1, 5.3.2, and 5.A.4, the sequence of policy updates (starting from θ^0 sufficiently close to a local optimum point θ^*) and Lagrange multipliers converges almost surely to a saddle point of the Lagrangian, i.e., $(\theta(k), \lambda(k)) \xrightarrow{a.s.} (\theta^*, \lambda^*)$. Then, θ^* is the local optimal solution of* (OPT-F).

5.B Practical PDPG Algorithm

A naive implementation of Algorithm 8 would result in a high-variance training procedure. In this section, we discuss several techniques for variance reduction, resulting in a more stable algorithm compared to the one proposed in Algorithm 8.

5.B.1 Step-wise KL-divergence Measure

In the policy distillation literature, some studies use a *trajectory-wise* KL-divergence (KL-F) as the distance metric [88], but the *step-wise* KL-divergence between the distribution is also common [30], which is defined as:

$$D_{KL}^{step}(\phi \parallel \theta) = \mathbb{E}_{x \sim d_{\pi_T}} \left[D_{KL}\big(\pi_T(\cdot|x;\phi) \parallel \pi_S(\cdot|x;\theta)\big) \right], \tag{5.36}$$

where

$$D_{KL}\big(\pi_T(\cdot|x;\phi) \parallel \pi_S(\cdot|x;\theta)\big) = \sum_{a\in\mathcal{A}} \pi_T(a|x;\phi) \log \frac{\pi_T(a|x;\phi)}{\pi_S(a|x;\theta)}. \tag{5.37}$$

In the next proposition, we explore the relations between these two methods.

Proposition 5.B.1. *The following relation holds between the trajectory-wise and step-wise KL-divergence metrics:*

$$D_{KL}(\phi \parallel \theta) \leq \mathbb{E}[H] \;\; D_{KL}^{step}(\phi \parallel \theta) \tag{5.38}$$

Proof. According to the definition of trajectory-wise KL-divergence, we have:

$$\begin{aligned}
D_{KL}(\mathbb{P}_\phi(\tau)||\mathbb{P}_\theta(\tau)) &= \sum_\tau \mathbb{P}_\phi(\tau) \log \frac{\mathbb{P}_\phi(\tau)}{\mathbb{P}_\theta(\tau)} \\
&= \sum_\tau \mathbb{P}_\phi(\tau) \log \frac{\mu(x_0)\prod_{t=0}^{H-1} \pi_T(a_t|x_t;\phi) P(x_{t+1}|x_t,a_t)}{\mu(x_0)\prod_{t=0}^{H-1} \pi_S(a_t|x_t;\theta) P(x_{t+1}|x_t,a_t)} \\
&= \sum_\tau \mathbb{P}_\phi(\tau) \sum_{t=0}^{H-1} \log \frac{\pi_T(a_t|x_t;\phi)}{\pi_S(a_t|x_t;\theta)} \\
&= \sum_{x\in\mathcal{X}, a\in\mathcal{A}} \sum_\tau \mathbb{P}_\phi(\tau) \sum_{t=0}^{H-1} \mathbb{I}_t(\tau;x,a) \log \frac{\pi_T(a|x;\phi)}{\pi_S(a|x;\theta)} \\
&\leq \sum_{x\in\mathcal{X}, a\in\mathcal{A}} \mathbb{E}[H] d_{\pi_T}(x) \pi_T(a|x;\phi) \log \frac{\pi_T(a|x;\phi)}{\pi_S(a|x;\theta)} \\
&= \mathbb{E}[H] \sum_{x\in\mathcal{X}} d_{\pi_T}(x) \sum_{a\in\mathcal{A}} \pi_T(a|x;\phi) \log \frac{\pi_T(a|x;\phi)}{\pi_S(a|x;\theta)} \\
&= \mathbb{E}[H] \; \mathbb{E}_{x\sim d_{\pi_T}} \big[D_{KL}(\pi_T(\cdot|x;\phi)||\pi_S(\cdot|x;\theta))\big]
\end{aligned}$$

Here, $\mathbb{I}_t(\tau;x,a)$ is the indicator of whether $(x_t = x, a_t = a)$ occurs along trajectory τ. Also, $d_{\pi_T}(x)$ is the distribution of being in state x under policy π_T, defined as

$$d_{\pi_T}(x) = \sum_{t=0}^{H_{max}} d_{t,\pi_T(x)}/H_{max},$$

and $d_{t,\pi_T(x)}$ is the probability of being in x at time t under policy π_T. □

According to this proposition, the step-wise KL distances can be used to provide an upper bound on the trajectory-wise one. In other words, if the step-wise KL multiplied by the expected horizon length is less than δ, then it is also correct for the trajectory-wise one.

The only remaining issue is that computing the expectation in (5.36) is not straightforward, since we only have access to the sample trajectories of the student during training. Using student samples to approximate the KL-divergence introduces some bias. One can alleviate this bias by incorporating importance sampling (IS) weights as

$$D_{KL}^{step}(\phi \parallel \theta) = \mathbb{E}_{x \sim d_{\pi_S}} \left[\frac{d_{\pi_T}(x)}{d_{\pi_S}(x)} D_{KL}\big(\pi_T(\cdot|x;\phi) \parallel \pi_S(\cdot|x;\theta)\big) \right]; \tag{5.39}$$

however, computing the stationary distributions is still a challenging task, even in simple MDPs with finite state space. One can follow the instructions of [55] for computing the correction values, but they add extra complications and are not the focus of this work. Even though we can more easily compute an (unbiased) estimate of reverse KL-divergence, we will utilize a biased estimation of the forward KL-divergence in most of our numerical analysis because of its "mean-seeking" property. Defining this biased forward KL-divergence is common in the literature, e.g., in [76].

Next, we illustrate with an example the low variance of the step-wise approximators compared to the trajectory-wise one.

Example: KL Approximation Accuracy using Full Information We design a simple 2×2 GridWorld example, as illustrated in Figure 5.B.1, to visualize the effect of approximating KL-divergence using Monte Carlo sampling. There is one agent in the top-left corner of the grid and it should reach the goal state located in the bottom-left one. We kept the problem as simple as possible since we wanted to generate all possible trajectories for computing the exact KL-divergence. The length of the horizon for this game is 4, so the total number of possible trajectories is $4^4 = 256$. One may notice that some of these trajectories might fully overlap, but that is fine for the purpose of this experiment. In this experiment, we have used a linear function approximator (i.e., a neural network with no hidden layer) and a medium-sized neural network.

We train a teacher that produces the actions right, left, up, and down with probabilities

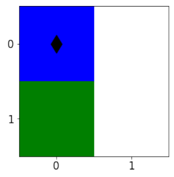
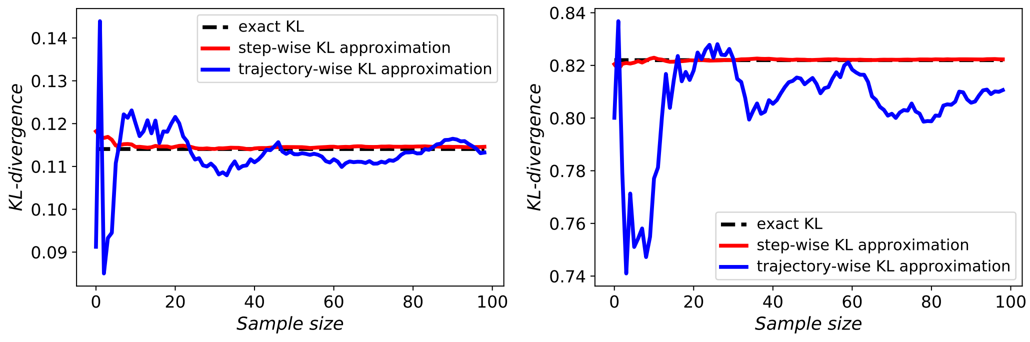


Figure 5.B.1 Illustration of the 2×2 GridWorld used for evaluating the effectiveness of KL approximations.

0.7, 0.0, 0.1, and 0.2, respectively. Once the trained network is available, we initialize the student's policy variables with those of the teacher plus a random number. Figure 5.B.2 shows the convergence behavior of the KL approximations to the exact value as we increase the Monte Carlo samples. The horizontal axis shows the number of sampled trajectories. As we observe, step-wise KL can provide a very good approximation of KL, even with a single trajectory sample, but the trajectory-wise approximation exhibits unstable behavior which is due to the intrinsic high variance of the estimator.



(a) Linear policy approximation

(b) Neural network function approximator with variables

Figure 5.B.2 Comparison of step-wise and trajectory-wise KL approximations, and their convergence to the exact KLs for two different policy approximators.

5.B.2 Practical PDPG Algorithm

According to the discussion of Section 5.4, we present the details of the practical PDPG algorithm in this section. We consider a setting in which the pre-trained teacher is readily available. The teacher articulates the status quo of solving the task. It can be a pre-trained RL agent itself, manually designed procedures, or a model of the teacher that has been trained using supervised learning from historical experiences. For example, the square-wave experiment uses handcrafted tabular policies, while in the wall leaping experiment, the teacher's policy—modeled with a neural network—is the outcome of an actor–critic algorithm. As long as we have cheap access to the teacher throughout the algorithm for numerous queries and get the corresponding probabilities for any given state and action pair, it is sufficient for our purposes.

Our approach is described in Algorithm 9. In every training iteration, we sample multiple trajectories under the student's policy, denoted by $\mathcal{T}^k$, which will be further utilized in approximating the policy gradient, KL approximations, and entropy. For more sample efficiency of the algorithm, we extract multiple sub-trajectories from each τ_j^k, and consider each sub-trajectory as an independent Monte Carlo sample. This is a common modification in policy gradient algorithms and can provide a satisfactory approximation from a single trajectory experience. Then the teacher provides an approximate probability for all actions at all visited states $x_{j,t}^k$. Once we know the probability of both student and teacher, we can compute the approximate step-wise KL-divergence from steps 8 and 9. Step 10 computes the entropy of student's current policy at each iteration.

Now, we have all approximations for computing update directions. In step 11, we use all previously computed sub-trajectory log-probabilities and their cumulative sampled reward along with the KL and entropy approximation to compute the loss. In this step, we also use a critic to provide a value of being at the initial point of each sub-trajectory $V(x_{j,t}^k)$, which will provide a baseline for variance reduction. Note that we didn't include the critic steps in our main algorithm since it follows a standard actor–critic design. Step 12 updates the policy parameters using the approximate gradient of loss with respect to θ at point θ^k. To be precise, the approximate gradient is employed in a first-order optimizer, e.g., ADAM [45], to update the θ values in the descent direction of the loss. Finally, the Lagrange multipliers λ and ζ are updated based on the amount of constraint violation at steps 13 and 14. We also periodically check to see whether λ^k has converged to λ_{max}, in which case we increase its quantity similar to Algorithm 8. Notice that since we have considered an equality constraint for entropy, its Lagrange multipliers can be positive or negative. To this end, we consider $\zeta \in [\zeta_{min}, \zeta_{max}]$ and if it converges to the boundary, we will increase the interval length.

Algorithm 9 Practical Primal-Dual Policy Gradient (PDPG) Algorithm for (OPT-F)

```
1: **input:** teacher's policy with weights $\phi$
2: **initialize:** student's policy with $\theta^0$, possibly equal $\phi$; initialize step size schedules $\alpha_1(\cdot)$, $\alpha_2(\cdot)$ and $\alpha_3(\cdot)$
3: **while** TRUE **do**
4:     **for** $k = 0, 1, \cdots$ **do**
5:         following policy $\theta^k$, generate a set of $N$ trajectories $\mathcal{T}^k = \{\tau_j^k,\ j = 1, 2, \cdots, N\}$, each starting from an initial state $x_0 \sim P_0(\cdot)$
6:         extract all trajectories $\tau_{j,t}^k$, which is a sub-trajectory of $\tau_j^k$ from $x_{j,t}^k$ onwards; also compute their corresponding accumulated reward $J(\tau_{j,t}^k)$ and log-probability $\log \tilde{\mathbb{P}}_\theta(\tau_{j,t}^k) := \sum_{t=0}^{H_j^k-1} \log \pi_S(a_{j,t}^k | x_{j,t}^k, \theta)$. Let $\bar{\mathcal{T}}^k$ be the set of all sub-trajectories for all visited states $x_{j,t}^k$
7:         query the teacher and compute $\pi_T(\cdot | x_{j,t}^k, \theta^k)$
8:         compute KL-divergence for all visited states $x_{j,t}^k$, i.e.,
               $$D_{KL}^{step}\big(\pi_T(\cdot|x_{j,t}^k;\phi) \parallel \pi_S(\cdot|x_{j,t}^k;\theta^k)\big) = \sum_{a\in\mathcal{A}} \pi_T(a|x_{j,t}^k;\phi) \log \frac{\pi_T(a|x_{j,t}^k;\phi)}{\pi_S(a|x_{j,t}^k;\theta^k)}, \quad \forall j, t \qquad (5.40)$$
9:         (***KL approximation***) compute the approximate KL-divergence as
               $$\hat{D}_{KL}^{step}(\phi \parallel \theta^k) = \frac{1}{N}\sum_{j=1}^{N} \frac{1}{H_j^k - 1} \sum_{t=0}^{H_j^k-1} \mathrm{CLIP}_\rho\Big(D_{KL}^{step}\big(\pi_T(\cdot|x_{j,t}^k;\phi) \parallel \pi_S(\cdot|x_{j,t}^k;\theta^k)\big)\Big) \qquad (5.41)$$
10:        (***entropy approximation***) compute the approximate entropy
               $$\hat{ent}(\theta^k) = -\frac{1}{N}\sum_{j=1}^{N} \frac{1}{H_j^k - 1} \sum_{t=0}^{H_j^k-1} \sum_{a\in\mathcal{A}} \pi_S(a|x_{j,t}^k;\theta^k) \log \pi_S(a|x_{j,t}^k;\theta^k) \qquad (5.42)$$
11:        (***compute loss***) compute the loss according to
               $$Loss(\theta^k, \lambda^k, \zeta^k) = \frac{1}{|\bar{\mathcal{T}}^k|} \sum_{\tau_{j,t}^k \in \bar{\mathcal{T}}^k} \log \tilde{\mathbb{P}}(\tau_{j,t}^k)(J(\tau_{j,t}^k) - V(x_{j,t}^k)) + \lambda^k(\hat{D}_{KL}^{step}(\phi \parallel \theta^k) - \delta) + \zeta^k(\hat{ent}(\theta^k) - \delta^{ent}) \qquad (5.43)$$
12:        (*$\theta$-**update***) update $\theta^k$ according to
               $$\theta^{k+1} = \Gamma_\Theta\Big[\theta^k - \alpha_1(k)\Big(\frac{1}{N}\sum_{j=1}^{N} \nabla_\theta Loss(\theta, \lambda^k, \zeta^k)\big|_{\theta=\theta^k}\Big)\Big] \qquad (5.44)$$
13:        (*$\lambda$-**update***) update $\lambda^k$ according to
               $$\lambda^{k+1} = \Gamma_\Lambda\Big[\lambda^k + \alpha_2(k)\Big(\hat{D}_{KL}^{step}(\phi \parallel \theta^k) - \delta\Big)\Big] \qquad (5.45)$$
14:        (*$\zeta$-**update***) update $\zeta^k$ with rule
               $$\lambda^{k+1} = \Gamma_Z\Big[\lambda^k + \alpha_3(k)\Big(\hat{ent}(\theta^k) - \delta^{ent}\Big)\Big] \qquad (5.46)$$
15:    **end for**
16:    update $\lambda_{max}$ similar to Algorithm 8
17:    **if** $\zeta^k$ converges to $\zeta_{max}$ **then**
18:        $\zeta_{max} \leftarrow \zeta_{max} + constant$
19:    **else if** $\zeta^k$ converges to $\zeta_{min}$ **then**
20:        $\zeta_{min} \leftarrow \zeta_{min} - constant$
21:    **else**
22:        return $\theta$, $\lambda$ and $\zeta$; break
23:    **end if**
24: **end while**
```

5.C Experiments Setup

In all of our experiments, the first step was to identify the teacher. In the square-wave experiment, we manually designed all teacher probabilities at every state. We also modeled situations in which the teacher is less "determined" and follows a more complicated decision-making scheme, such as in our wall leaping experiment, in which the teacher is the policy of an agent trained using the actor–critic algorithm.

Although we could have initialized the student's policy randomly, we chose to initialize it with a pre-trained neural network. In all experiments, we train the neural network for the unconstrained problem and using the actor–critic algorithm. Similarly, the student's critic is initialized from the previously trained critic. Notice that the student's initial policy does not need to be the same as the teacher's policy. For example, the initial student policy in the square-wave experiment always takes the horizontal path, which is totally different from that of the teacher. Nevertheless, starting from a policy close to the teacher would expedite the learning process since there is a high probability of finding an improved policy in the proximity of the teacher.

However, starting from a previously trained network can bring some difficulties. For example, having a deterministic initial policy would lead to a limited amount of exploration. To mitigate this issue, we use a temperature hyper-parameter when sampling from our softmax, similar to [8]. In this method, we normalize the output of the neural network—called *logits*—with temperature and then compute the sampling probabilities as $\pi(\cdot|s;\theta) = \text{softmax}(\textit{logits} \ / \ \textit{temperature})$. Using a temperature greater than 1 smoothes out the sampling probability distribution, so there will be a higher chance of visiting less-explored states.

In all of our experiments, we used a neural network with two hidden layers, each with 64 neurons. We used the ADAM optimizer [45] with step size 1e$-$3 to update the student's policy and critic. The temperature is 5; λ and ζ start from 1. The learning rate for λ and ζ starts from 1e$-$3 and decays to 1e$-$3 during training. The right hand sides of all entropy constraints are set to 0.02. We also have a plan to open-source our PyTorch code soon.

Bibliography

[1] Naoki Abe, Naval Verma, Chid Apte, and Robert Schroko. Cross channel optimized marketing by reinforcement learning. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 767–772. ACM, 2004.

[2] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 22–31. JMLR. org, 2017.

[3] Ivo Adan and Gideon Weiss. Exact fcfs matching rates for two infinite multitype sequences. *Operations research*, 60(2):475–489, 2012.

[4] Ivo Adan, Ana Bušić, Jean Mairesse, and Gideon Weiss. Reversibility and further properties of fcfs infinite bipartite matching. *Mathematics of Operations Research*, 43 (2):598–621, 2017.

[5] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study.* Princeton university press, 2006.

[6] Claudia Archetti and Maria Grazia Speranza. The split delivery vehicle routing problem: a survey. In *The vehicle routing problem: Latest advances and new challenges*, pages 103–122. Springer, 2008.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *In International Conference on Learning Representations*, 2015.

[8] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[9] Dimitri P Bertsekas. *Nonlinear programming.* Athena scientific Belmont, 1999.

[10] Dimitri P Bertsekas. *Convex optimization theory.* Athena Scientific Belmont, 2009.

[11] Shalabh Bhatnagar, Richard S Sutton, Mohammad Ghavamzadeh, and Mark Lee. Natural actor-critic algorithms. *Automatica*, 45(11), 2009.

[12] Vivek S Borkar. *Stochastic approximation: a dynamical systems viewpoint*, volume 48. Springer, 2009.

[13] Burak Büke and Hanyi Chen. Stabilizing policies for probabilistic matching systems. *Queueing Systems*, 80(1-2):35–69, 2015.

[14] Ana Bušić and Sean Meyn. Optimization of dynamic matching models. *arXiv preprint arXiv:1411.1044*, 2014.

[15] Ana Bušić, Varun Gupta, and Jean Mairesse. Stability of the bipartite matching model. *Advances in Applied Probability*, 45(2):351–378, 2013.

[16] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. In *Innovations in multi-agent systems and applications-1*, pages 183–221. Springer, 2010.

[17] René Caldentey, Edward H. Kaplan, and Gideon Weiss. Fcfs infinite bipartite matching of servers and customers. *Advances in Applied Probability*, 41(3):695–730, 2009.

[18] Kan Chen, Jiang Wang, Liang-Chieh Chen, Haoyuan Gao, Wei Xu, and Ram Nevatia. Abc-cnn: An attention based convolutional neural network for visual question answering. *arXiv preprint arXiv:1511.05960*, 2015.

[19] Shi-Yong Chen, Yang Yu, Qing Da, Jun Tan, Hai-Kuan Huang, and Hai-Hong Tang. Stabilizing reinforcement learning in dynamic environment with application to online

recommendation. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1187–1196. ACM, 2018.

[20] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *Conference on Empirical Methods in Natural Language Processing*, 2014.

[21] Yinlam Chow, Mohammad Ghavamzadeh, Lucas Janson, and Marco Pavone. Risk-constrained reinforcement learning with percentile risk criteria. *Journal of Machine Learning Research*, 18(167):1–167, 2017.

[22] Yinlam Chow, Ofir Nachum, Edgar Duenez-Guzman, and Mohammad Ghavamzadeh. A lyapunov-based approach to safe reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 8092–8101, 2018.

[23] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[24] Geoff Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

[25] Harold Cramer. Mathematical methods of statistics, princeton univ. *Press, Princeton, NJ*, 1946.

[26] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, pages 2702–2711, 2016.

[27] Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in Neural Information Processing Systems*, 2017.

[28] Shuo Feng, Peyman Setoodeh, and Simon Haykin. Smart home: Cognitive interactive people-centric internet of things. *IEEE Communications Magazine*, 55(2):34–39, 2017.

[29] Ricardo Fukasawa, Humberto Longo, Jens Lysgaard, Marcus Poggi de Aragão, Marcelo Reis, Eduardo Uchoa, and Renato F Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical programming*, 106 (3):491–511, 2006.

[30] Dibya Ghosh, Avi Singh, Aravind Rajeswaran, Vikash Kumar, and Sergey Levine. Divide-and-conquer reinforcement learning. *arXiv preprint arXiv:1711.09874*, 2017.

[31] Joren Gijsbrechts, Robert N Boute, Jan A Van Mieghem, and Dennis Zhang. Can deep reinforcement learning improve inventory management? performance and implementation of dual sourcing-mode problems. *Performance and Implementation of Dual Sourcing-Mode Problems (December 17, 2018)*, 2018.

[32] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[33] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[34] Fred Glover and Manuel Laguna. Tabu search*. In *Handbook of combinatorial optimization*, pages 3261–3362. Springer, 2013.

[35] Bruce L Golden, Subramanian Raghavan, and Edward A Wasil. *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43. Springer Science & Business Media, 2008.

[36] Inc. Google. Google's optimization tools (or-tools), 2018. URL `https://github.com/google/or-tools`.

[37] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396, 2017.

[38] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016. URL `http://www.gurobi.com`.

[39] Itai Gurvich and Amy Ward. On the dynamic control of matching queues. *Stochastic Systems*, 4(2):479–523, 2014.

[40] Seunghoon Hong, Junhyuk Oh, Honglak Lee, and Bohyung Han. Learning transferrable knowledge for semantic segmentation with deep convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3204–3212, 2016.

[41] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[42] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. 2015.

[43] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *arXiv preprint arXiv:1708.07902*, 2017.

[44] BRK Kashyap. The double-ended queue with bulk service and limited waiting space. *Operations Research*, 14(5):822–834, 1966.

[45] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Machine Learning*, 2015.

[46] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[47] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[48] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems*, pages 1008–1014, 2000.

[49] Gilbert Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European journal of operational research*, 59(3):345–358, 1992.

[50] Gilbert Laporte, Michel Gendreau, Jean-Yves Potvin, and Frédéric Semet. Classical and modern heuristics for the vehicle routing problem. *International transactions in operational research*, 7(4-5):285–300, 2000.

[51] Jan Leike, Miljan Martic, Victoria Krakovna, Pedro A Ortega, Tom Everitt, Andrew Lefrancq, Laurent Orseau, and Shane Legg. Ai safety gridworlds. *arXiv preprint arXiv:1711.09883*, 2017.

[52] Jan Karel Lenstra and AHG Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227, 1981.

[53] Xiujun Li, Lihong Li, Jianfeng Gao, Xiaodong He, Jianshu Chen, Li Deng, and Ji He. Recurrent reinforcement learning: a hybrid approach. *arXiv preprint arXiv:1509.03044*, 2015.

[54] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *International Conference on Learning Representations*, 2016.

[55] Qiang Liu, Lihong Li, Ziyang Tang, and Dengyong Zhou. Breaking the curse of horizon: Infinite-horizon off-policy estimation. In *Advances in Neural Information Processing Systems*, pages 5361–5371, 2018.

[56] László Lovász and Michael D Plummer. *Matching theory*, volume 367. American Mathematical Soc., 2009.

[57] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *Conference on Empirical Methods in Natural Language Processing*, 2015.

[58] Jean Mairesse and Pascal Moyal. Stability of the stochastic matching model. *Journal of Applied Probability*, 53(4):1064–1077, 2016.

[59] Aranyak Mehta. Online matching and ad allocation. *Theoretical Computer Science*, 8(4):265–368, 2012.

[60] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[61] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[62] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[63] Nasser M Nasrabadi. Pattern recognition and machine learning. *Journal of Electronic Imaging*, 16(4):049901, 2007.

[64] Mohammadreza Nazari and Alexander L Stolyar. Reward maximization in general dynamic matching systems. *Queueing Systems*, 91(1-2):143–170, 2019.

[65] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takac. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9861–9871, 2018.

[66] Graham Neubig. Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*, 2017.

[67] Afshin Oroojlooyjadid, Mohammadreza Nazari, Lawrence Snyder, and Martin Takáč. A deep q-network for the beer game with partial information. *arXiv preprint arXiv:1708.05924*, 2017.

[68] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[69] Ismail Parsa and Ken Howes. Kdd cup 1998 data, 1999. data retrieved from The UCI KDD Archive, https://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html.

[70] Edwin Pednault, Naoki Abe, and Bianca Zadrozny. Sequential cost-sensitive decision making with reinforcement learning. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 259–268. ACM, 2002.

[71] Erica L Plambeck and Amy R Ward. Optimal control of a high-volume assemble-to-order system with maximum leadtime quotation and expediting. *Queueing Systems*, 60(1):1–69, 2008.

[72] Ulrike Ritzinger, Jakob Puchinger, and Richard F Hartl. A survey on dynamic and stochastic vehicle routing problems. *International Journal of Production Research*, 54 (1):215–231, 2016.

[73] Andrei A Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

[74] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. Computational capabilities of graph neural networks. *IEEE Transactions on Neural Networks*, 20(1):81–102, 2009.

[75] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6):233–242, 1999.

[76] Simon Schmitt, Jonathan J Hudson, Augustin Zidek, Simon Osindero, Carl Doersch, Wojciech M Czarnecki, Joel Z Leibo, Heinrich Kuttler, Andrew Zisserman, Karen Simonyan, et al. Kickstarting deep reinforcement learning. *arXiv preprint arXiv:1803.03835*, 2018.

[77] Jennie Si and Yu-Tsung Wang. Online learning control by association and reinforcement. *IEEE Transactions on Neural networks*, 12(2):264–276, 2001.

[78] David Silver, Leonard Newnham, David Barker, Suzanne Weller, and Jason McFall. Concurrent reinforcement learning from customer interactions. In *International Conference on Machine Learning*, pages 924–932, 2013.

[79] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[80] Jean-Jacques E Slotine, Weiping Li, et al. *Applied nonlinear control*, volume 199. Prentice hall Englewood Cliffs, NJ, 1991.

[81] Lawrence V Snyder and Zuo-Jun Max Shen. *Fundamentals of Supply Chain Theory*. John Wiley & Sons, 2nd edition, 2018.

[82] Alexander L Stolyar. Maximizing queueing network utility subject to stability: Greedy primal-dual algorithm. *Queueing Systems*, 50(4):401–457, 2005.

[83] Alexander L Stolyar. Greedy primal-dual algorithm for dynamic resource allocation in complex networks. *Queueing Systems*, 54(3):203–220, 2006.

[84] Alexander L Stolyar and Tolga Tezcan. Control of systems with flexible multi-server pools: a shadow routing approach. *Queueing Systems*, 66(1):1–51, 2010.

[85] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[86] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[87] Aviv Tamar, Dotan Di Castro, and Shie Mannor. Policy gradients with variance related risk criteria. In *Proceedings of the twenty-ninth International Conference on Machine Learning*, pages 387–396, 2012.

[88] Yee Teh, Victor Bapst, Wojciech M Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4496–4506, 2017.

[89] Georgios Theocharous, Philip S Thomas, and Mohammad Ghavamzadeh. Ad recommendation systems for life-time value optimization. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1305–1310. ACM, 2015.

[90] Georgios Theocharous, Philip S Thomas, and Mohammad Ghavamzadeh. Personalized ad recommendation systems for life-time value optimization with guarantees. In *IJCAI*, pages 1806–1812, 2015.

[91] Andrea Lockerd Thomaz, Cynthia Breazeal, et al. Reinforcement learning with human teachers: Evidence of feedback and guidance with implications for learning performance. In *Aaai*, volume 6, pages 1000–1005. Boston, MA, 2006.

[92] Yegor Tkachenko. Autonomous crm control via clv approximation with deep reinforcement learning in discrete and continuous action space. *arXiv preprint arXiv:1504.01840*, 2015.

[93] Yegor Tkachenko, Mykel J Kochenderfer, and Krzysztof Kluza. Customer simulation for direct marketing experiments. In *Data Science and Advanced Analytics (DSAA), 2016 IEEE International Conference on*, pages 478–487. IEEE, 2016.

[94] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem.* SIAM, 2002.

[95] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. *arXiv preprint arXiv:1804.02477*, 2018.

[96] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[97] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. 2016.

[98] Christos Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *European journal of operational research*, 113(2): 469–499, 1999.

[99] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

[100] Ronald J Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

[101] Anthony Wren and Alan Holliday. Computer scheduling of vehicles from one or more depots to a number of delivery points. *Operational Research Quarterly*, pages 333–344, 1972.

[102] Tianjun Xiao, Yichong Xu, Kuiyuan Yang, Jiaxing Zhang, Yuxin Peng, and Zheng Zhang. The application of two-level attention models in deep convolutional neural network for fine-grained image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 842–850, 2015.

[103] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

Biography

Mohammadreza Nazari is a Ph.D. candidate in the Department of Industrial and Systems Engineering at Lehigh University in Bethlehem, PA. After earning his undergraduate degree in Industrial Engineering, he ranked 1st among more than 8,000 Industrial Engineering Participants in the Iranian National Graduate Qualification Exam and was awarded admission with honor to the Masters degree program at Sharif University of Technology. After several years of experience in industry as an ERP consultant and Business Process Analyst, he joined Lehigh University to pursue his Ph.D. with Professors Lawrence Snyder and Martin Takáč. During his Ph.D. studies, he has been engaged in analyzing and designing novel Machine Learning and Reinforcement Learning solutions for real-world problems in Supply Chain and Marketing. He has been collaborating with SAS Institute Inc. over the last two years, with a desire to build practical solutions that incorporate Artificial Intelligence. Because of his excellence in research and service, he was twice awarded the *ISE Ph.D. Student of Year title* (2017, 2019), and *Graduate Student Leadership and Service* prize, 2017.