Learning Dynamics and Reinforcement in Stochastic Games

by

John Holler

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Mathematics)
in The University of Michigan

John Holler

johnholl@umich.edu

orcid.org/0000-0002-5407-7041

Dedication

This dissertation is dedicated to Virginia Kunisch.

Acknowledgments

The author thanks Haley Hofbauer, Mishaal Muqaddam, Alex Gross, Joe Kraisler, Kristen Holler, Ed and Karen Holler, Igor Kriz, Satinder Singh, Tony Qin, Xiaocheng Tang, Yan Jiao, Martin Strauss, David Leslie

Chapters

List of Figures

Chapter 1

Q-Learning Approaches to Dynamic Multi-Driver Dispatching and Repositioning

The driver management system at a ride-sharing company must make decisions both for assigning available drivers to nearby unassigned passengers (hereby called orders) over a large spatial decision-making region (e.g., a city) and for repositioning drivers who have no nearby orders. Such decisions not only have immediate to short-term impact in the form of revenue from assigned orders and driver availability, but also long-term effects on the distribution of available drivers across the city. This distribution critically affects how well future orders can be served. The need to address the exploration-exploitation dilemma as well as the delayed consequences of assignment actions makes this a Reinforcement Learning (RL) problem.

Recent works [**?**, **?**] have successfully applied Deep Reinforcement Learning techniques to dispatching problems, such as the Traveling Salesman Problem (TSP) and the more general Vehicle Routing Problem (VRP), however they have primarily focused on *static* (i.e., those where all orders are known up front) and/or *single-driver* dispatching problems. In contrast, for use in ride-sharing applications, one needs to find good policies that can accommodate a *multi-driver* dispatching scheme where demands are not known up front but rather generated *dynamically* throughout an episode (e.g., a day). We refer to this problem as a multi-driver vehicle dispatching and repositioning problem (MDVDRP).

We define an MDVDRP as a continuous-time semi-Markov decision process with the following state-reward dynamics. At any time, state is given by a set of requesting *orders* $o_t^i \in \mathcal{O}_t$, a set of drivers *drivers* $d_t^i \in \mathcal{D}_t$, and time of day t. There is also a set of repositioning movements $m^j \in \mathcal{M}$, which are not part of state but will be part of the action space. The size of $\mathcal{O}_t$ will change in time due to orders stochastically appearing in the environment and disappearing as they are served or canceled. Orders are canceled if they do not receive a driver assignment within a given time window. $\mathcal{D}_t$ can be further subdivided into *available drivers* and *unavailable drivers*. A driver is unavailable if and only if they are currently fulfilling an order. An action is a pairing of an available driver with either a requesting order or repositioning movement. An order is characterized by a pickup location (where the passenger is located), and end location (where the customer wants to go), a price, and the amount of time since the order arrived in the system. A driver is described by her position if she is available, and her paired order or reposition movement if she is unavailable. A reposition movement is described by a direction and duration, e.g. "Travel west for three minutes". When a driver is assigned to an order, the agent receives a reward equal to the price of that order, and the driver becomes unavailable until after it has picked up its order at the start location and dropped it off at the end location. When a driver is assigned a repositioning movement, the agent receives no reward and repositions until it either reaches the maximum repositioning duration or is assigned an order. When any action is taken in the environment, time advances to the next time that there is a driver that is available and not repositioning. Note that if there are multiple non-repositioning drivers at time t and multiple requesting orders, then time will not advance after the first action is taken since there will still be at least one available non-repositioning driver and order pairing.

Heuristic solutions construct an approximation to the true problem by ignoring the spatial extent, or the temporal dynamics, or both, and solve the approximate problem exactly. One such example is myopic pickup distance minimization (MPDM), which ignores temporal dynamics and always assigns the closest available driver to a requesting order [**?**] We illustrate this below in two simple dispatching domains that capture the essence of these suboptimalities, and demonstrate that our methods can overcome such issues.

In this paper we construct a global state representation along with a neural network (NN) architecture that can take this global state as input and produce action-values (Q-values). Due to the variable number of orders, which appear both as part of state and part of actions, we make use of attention mechanisms both for input and output of the NN. We then present two methods for training this network to perform dispatching and repositioning: single-driver deep Q-learning network (SD-DQN) and multi-driver deep Q-learning network (MD-DQN). Both approaches are based on the deep Q-learning network (DQN) algorithm [**?**], but differ from each other in the extent to which individual driver behaviors are coordinated. SD-DQN learns a Q-value function of global state for single drivers by accumulating rewards along *single-driver* trajectories. On the other hand, MD-DQN uses *system-centric* trajectories, so that the Q-value function accumulates rewards for all drivers. We find that MD-DQN can learn superior behavior policies in some cases, but that in practice SD-DQN is competitive in all environments and scales well with the number of drivers in the MDVDRP while MD-DQN performs poorly in real-data domains. Empirically we compare performance of SD-DQN and MD-DQN on a static assignment problem, illustrative MDVDRPs, and a data-driven MDVDRP designed using real world ride-sharing dispatching data. There have been several recent approaches to solving dispatching and routing problems. Some examples include Bello et al. ([**?**]), Nazari et al. ([**?**]), and Vinyals et al. ([**?**]). All of these works use an encoding-decoding scheme, where first information is processed into a global-context, and then from this context actions are decoded sequentially. Pointer networks [**?**] offer an approximate solution to TSPs by encoding cities (in our terminology, orders) sequentially with a recurrent network, and then producing a solution by sequentially "pointing" to orders using an attention mechanism [**?**]. The network is trained with a supervised loss function by imposing a fixed ordering rule on decoded orders. Bello et al. ([**?**]) build off of the pointer network architecture, but train the network with policy gradients and so may dispense with the fixed ordering during the decoding phase.

While related to our work, the above papers typically focus on problems that are *static* and/or *single-driver* (in the sense that there is only one driver that must be controlled). In contrast, we are interested in making dispatching decisions for many drivers in a dynamic environment. Furthermore, the sequential encoding in Bello et al. and Vinyals et al. introduces an unnecessary forced ordering of inputs into the neural network by encoding using a recurrent network. Instead we follow an architecture more closely related to Nazari et al. ([**?**]), which uses an attention mechanism [**?**] for encoding, however they only apply their architecture to static and single-driver vehicle routing problems. We depart from their architecture further by dispensing with the recurrent network used for decoding. Instead, we construct a global representation of state that is encoded and decoded in a completely feed-forward fashion.

Next we discuss a previous work on a problem more closely related to MDVDRPs. Oda et al. [**?**] offer a *value-based* approach to the dynamic fleet management problem, which is a strict subproblem of the MDVDRP. In dynamic fleet management, one is concerned only with *repositioning* available drivers, while driver-order assignments are handled via hard-coded rules (in their case, minimizing pickup distance). A MDVDRP combines fleet management with driver-order matching.

Another thread of related work comes from the multi-agent reinforcement learning literature. Specifically, our two training approaches, SD-DQN and MD-DQN, are analogous to the multi-agent reinforcement learning (MARL) algorithms of "independent Q-learning" [**?**] and "team-Q learning" [**?**] respectively. The tradeoffs between these two approaches to team problems are broadly known but not well understood. Team-Q has the benefit of theoretical justification, but its action set grows exponentially in the number of agents. On the other hand, while mostly lacking in theory, independent Q-learning has been leveraged successfully in a number of problems [**?**, **?**]. Claus and Boutilier ([**?**]) conjecture that independent Q-learners will converge to a Nash equilibrium under appropriate assumptions in team games, however this claim remains unproven. Our results reflect the theme that SD-DQN works well in practice at a variety of scales in team problems despite the dearth of convergence results, while MD-DQN has considerable difficulty scaling beyond a small number of agents.

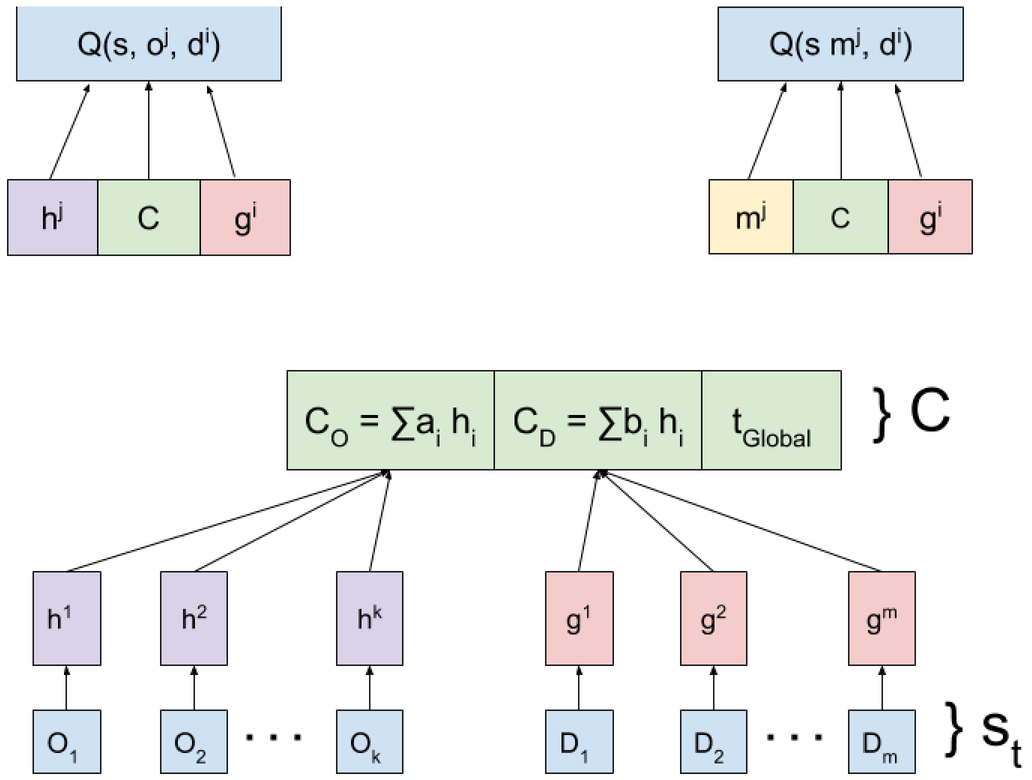

Figure 1.1: The neural network architecture used in SD-DQN and MD-DQN. Order and driver embeddings are attended to for global context. The network then uses global context to query order/driver pairs (represented by the matrix in the top left) and reposition/driver pairs (represented by the matrix in the top right) to produce Q-values for all possible actions.

1.0.1 Neural Network Overview

At time t there is a collection of orders $o_t^i \in \mathcal{O}_t$, drivers $d_t^j \in \mathcal{D}_t$, and subset $\mathcal{D}_t^{avail} \subset \mathcal{D}_t$ of available drivers. A driver is available if it is not currently serving an order. State is given to the neural network as $s_t = (\mathcal{O}_t, \mathcal{D}_t, \mathcal{D}_t^{avail}, t)$. At time t, the action space is made up of available driver-order pairings or non-repositioning driver-repositioning pairings.

Input representation

The exact vector representation of o_t^i and d_t^j depend on the environment. In the static assignment problem [**?**], we are only concerned with the position of drivers and *start positions* of orders. Therefore drivers and orders are each represented as two-dimensional vectors of their x and y coordinates. In MDVDRP, orders are given as six-dimensional vectors consisting of starting x position, starting y position, ending x position, ending y position, price, and time waiting, where time waiting is the difference between the current time and the time that the order first began requesting a driver. A driver is represented by a six-dimensional vector: an x position, y position, time to completion, repositioning x and y coordinates, and reposition counter. If the driver is available, its x and y position are given by its actual location, and time to completion is 0. If the driver is not available, the x and y position are the ending location of the order it is servicing, and the time to completion is the time it will take to finish the order. If a driver is repositioning, the direction of repositioning is reflected in the repositioning coordinates, and reposition counter counts down from the maximum repositioning time.

Embedding and global context

The network first embeds orders $\{o_t^i\}_i$ and drivers $\{d_t^j\}_j$ into memory cells $\{h_t^i\}_i$ and $\{g_t^j\}_j$ respectively found in the purple and red boxes in **??**. Then, a single round of attention is performed over each to produce a *global order context* C_t^O and *global driver context* C_t^D. These contexts as well as global time are concatenated to produce a *global context vector* C_t^G, which is the system's internal global representation of state.

Action-value calculation

The network then computes two types of Q-values: those for driver-order pairs and those for driver-reposition pairs. Each of these two processes can be viewed as their own attention mechanisms over pairs, with the attention coefficients interpreted as Q-values. More precisely, for available driver-order pairs we construct a small fully connected neural network Att_o such that $Q(s_t, d_t^j, o_t^i) = Att_o(C_t, g_t^j, h_t^i)$. Similarly, for available driver-reposition pairs, we construct another small neural network Att_r, so that $Q(s_t, d_t^j, m_t^i) = Att_r(C_t, g_t^j, m_t^i)$ where m_t^i is a vector representation of a reposition action. The top left of **??** represents a generic example of Att_o while the top right represents Att_r. In all repositioning experiments, there are 9 reposition actions consisting of 8 cardinal directions and a stationary action (no movement). The m_t^i's are represented as one-hot vectors, and displayed as yellow boxes in **??**.

1.0.2 Single-driver and multi-driver Q-values

We take two approaches to training the above network: single-driver DQN (SD-DQN) and multi-driver DQN (MD-DQN). Each can be viewed as a 1-step bootstrapping approach like in the standard DQN, but they differ from one another in the form of one step data that is given to the network. As a result, the learned Q-values in each approach have distinct semantics.

In SD-DQN, we use *driver-centric* transitions. At time t, the system is in global state $s_t = (O_t, \mathcal{D}_t, \mathcal{D}_t^{avail}, t)$ and a driver-order or driver-reposition action is selected, yielding reward r_t. Let d be the selected driver, and a_t denote the action. We then proceed to make dispatching and repositioning decisions for other drivers as they become available, until eventually driver d is available again in state $s_{t'} = (O_{t'}, \mathcal{D}_{t'}, \mathcal{D}_{t'}^{avail}, t')$. The change in time $t' - t$ is the time it takes for driver d to complete a single trip or repositioning, which is typically between 10 and 30 minutes for trips or $2 - 3$ minutes for repositionings. In SD-DQN, this yields a transition $(s_t, a_t, r_t, s_{t'})$. To train our network in the SD-DQN setting, we update the outputs of the network using the target:

$$\widehat{Q}(s_t, a_t; \theta_t) = r_t + \gamma^{t'-t} \cdot max_{a'} Q^T(s_{t'}, a'; \theta_t'), \tag{1.1}$$

where θ_t are the current network weights and θ'_t are the weights of the target network, which are slow updating weights that sync with the network's current weights at fixed intervals. To account for the fact that transitions occur over a variable time interval, future value is discounted continuously with discount factor γ. The network is trained to reduce the Huber loss between $\widehat{Q}(s_t, a_t)$ and $Q(s_t, a_t)$. Intuitively, SD-DQN updates the network towards driver-centric Q-values, only accounting for the discounted return associated to a single driver. We collect driver-centric transitions from each driver into a single shared replay buffer. The drawback of this method is that, as we learn the single driver Q-values, we update the behavior of *all drivers*, and so one perspective is that the driver, as an agent, is learning in a nonstationary MDP.This has the potential to cause instability issues in training, though we did not observe such issues in our experiments. Another issue with SD-DQN is that it may train single drivers to behave selfishly. That is, drivers might learn to choose actions that are best for themselves rather than actions that promote the greatest overall reward for the system. This issue is raised in work on learning in collectives [**?**, **?**], however the interaction of individual learning and system optimization is not well understood.

In MD-DQN we use *system-centric* transitions. At time t, the system is in global state $s_t = (\mathcal{O}_t, \mathcal{D}_t, \mathcal{D}_t^{avail}, t)$ and we select action a_t, yielding reward r_t. When the next new driver becomes available, we transition to state $s_{t'} = (\mathcal{O}_{t'}, \mathcal{D}_{t'}, \mathcal{D}_{t'}^{avail}, t')$. In contrast to the SD-DQN transition, the change in time $t' - t$ is the time it takes for the next available driver to appear, which is on the order of fractions of a second in large cities. As in the SD-DQN case, this yields a 1-step transition $(s_t, a_t, r_t, s_{t'})$. We use the same target and update procedure from equation (1), but with this different transition as input data. MD-DQN will update the network towards a global, system-centric Q-value that sums the discounted rewards of all drivers together. This means the Q-values produced by MD-DQN will be approximately n times larger than those of SD-DQN, where n is the number of drivers. Ignoring issues related to function approximation MD-DQN provides a correct learning signal for solving an MDVDRP. However, as a practical matter, the Q-values learned in MD-DQN are based on many more transitions, and therefore should be harder to learn. This has been our empirical experience. In the results section we describe steps we needed to take in order to stabilize MD-DQN learning, but the primary change we make is to do *n*-step Q-learning [**?**] with *n* equal to the number of drivers in the system.

1.0.3 Architecture details

For ease of exposition, in the following section we omit subscript *t*'s that had denoted time in previous sections.

Memory embedding. We begin with a global state *s*, consisting of order and drivers and time. All orders and drivers are both embedded into length 128 memory cells using a single layer network with 128 output units followed by RELU nonlinearities.

Attention for global context. Given a set of N order memories h^i and M driver memories g^j. The attention mechanism for orders/drivers are given by the following equations:

$$C^O = \sum_{i=1}^{N} a_i h^i, \quad C^D = \sum_{j=1}^{M} b_j g^j \tag{1.2}$$

where

$$a_i = \sigma(v_o \cdot tanh(W^O \cdot h^i)), b_j = \sigma(v_d \cdot tanh(W^D \cdot g^j)) \tag{1.3}$$

and σ is a sigmoid activation function, W^O and W^D are 128 dimensional square matrices, and v^O and v^D are 128 dimensional vectors so that a_i and b_j are scalars. Both W's and v's are trained.

We concatenate the two contexts, along with the episode time t, to produce a 257 dimensional global context vector

$$C_G = [C_o|C_d|t] \tag{1.4}$$

Q-values. To compute a driver-order Q-value $Q(s, d^j, o^i)$, we concatenate the global context with the order's memory embedding h^i and driver's memory embedding g^j, and pass this through a fully connected layer with 64 units and RELU activation, and finally pass this to a single linear unit. We use the same network architecture for driver-repositioning pairs, but we do not share weights between the driver-order network and driver-repositioning network.

Further training details

In all experiments, we use a replay memory that is initialized with random behavior transitions. The size of replay memory as well as how frequently the target network is synced are both environment dependent, and are specified in the appendix. We also use a target network for setting Q-value targets. During training, each training loop begins by taking one step in the environment. Behavior is ϵ-greedy with respect to the network's Q-values, where ϵ is linearly annealed in all experiments as specified in the appendix. For both SD-DQN and MD-DQN, one new transition will be added to replay memory (though they differ in what this one step transition is). Then, we sample a batch of 32 transitions from replay memory, and perform a Q-value update using equation (1). For all experiments, $\gamma = 0.99$. Gradient's are applied using the Tensorflow implementation of RMSProp with gradients clipped at 100.

Results are presented in table format. Entries with error bars are computed as follows. We run the learning method (SD-DQN or MD-DQN) 4 times, with each experiment differing only in random seed. For each run, once learning has appeared to converge, we average reward, pickup distance, and orders served percentages over 20 episodes. We then compute the average and error bars across the four runs. If no error bars are included, this means the table is showing results over a single run, with entries averaged over 20 episodes.

1.0.4 Static multi-driver assignment problem

The assignment problem [**?**] is a combinatorial optimization problem defined by a *cost matrix*, where the ij^{th} entry represents the cost of assigning the i^{th} row object to the j^{th} column object. The goal is to produce an assignment that minimizes the sum of costs. In the context of driver assignment, the assignment cost is given by the Euclidean distance between order o^i and driver d^j. The assignment problem is the core subproblem for a dispatching problem with fixed windowing and a distance-minimization objective. An episode is initialized by a uniform random distribution of k orders and k drivers over the unit square. At each step, we choose a random driver, and the agent selects an order to match with the given driver. The reward associated to this action is the negative Euclidean distance between the driver-order pair. We do not perform discounting due to the static

policy	total pickup distance
Random	10.25
SD-DQN	4.83 ± .06
MD-DQN	4.12 ± .03
Optimal	3.82

Table 1.1: 20 driver 20 order static assignment problem

nature of the problem.

The assignment problem is a particularly good environment for demonstrating the potential miscoordination of SD-DQN. For the single-driver approach, each transition ends in a terminal state, meaning that Q-learning reduces to one step reward prediction. Therefore a policy which is greedy with respect to single driver Q-values will be suboptimal since it does not learn any coordination between drivers. On the other hand, an MD-DQN agent is concerned with maximizing the aggregate return across all drivers, and so should be capable of learning a better policy.

Results are summarized in **??**. We show total distance traveled (that is, the sum of the distances of all assignments) for SD-DQN and MD-DQN when there are 20 orders and 20 drivers. We compare them to optimal solutions as well as *Random assignments*. The results reflect our intuition regarding the shortcomings of SD-DQN in sensitive coordination problems. Namely, SD-DQN performs quite a bit worse than MD-DQN. This emphasizes the innate desirability of MD-DQN – it is capable of learning more complex coordination behavior.

1.0.5 Dynamic Multi-Driver Dispatching Problems

The remaining experiments will focus on *dynamic* dispatching problems. In the dynamic setting, orders arrive at different times throughout an episode. Additionally, we are focused on ride-sharing, so orders now are defined by pickup *and* dropoff locations. When a driver is assigned to an order, it must first navigate to the pickup location and then travel to the dropoff location. We first present results on small domains where myopic policies are demonstrably suboptimal. Then, we apply the SD-DQN and MD-DQN approaches to a large-scale dispatching simulator derived from real-world data collected from DiDi Chuxing. We find that SD-DQN outperforms all other methods in the realistic simulators, but is occasionally worse than MD-DQN in illustrative domains.

Illustrative domains with no repositioning

In this group of experiments, we use a simple dispatching simulator to show that both SD- and MD-DQN can learn good policies in two key scenarios where myopic behavior fails. The city is represented by the unit square. In these domains, at the start of a new episode, drivers are all located at a "dispatching depot" at position $[0.5, 0.5]$. Drivers travel at a constant speed of .1, and travel along straight lines from their initial position to the order pickup location, and then from order pickup to order dropoff location. Order arrivals are generated according to a Poisson distribution, with controllable parameter κ. In the following experiments, κ is set to either 3 or 10 (that is, average order arrivals per unit time are either 3 or 10) to simulate "low demand" and "high demand" environments. The pickup and dropoff locations as well as the reward for an order are specified below for two different environment settings. An episode lasts 5000 timesteps.

Surge domain

The Surge domain illustrates an explicit, temporal effect caused by order pricing that cannot be exploited by myopic dispatchers. In the Surge domain, there are three regions: left, center, and right. One quarter of all orders go from the center to the upper-left region, one quarter from center to bottom-right, one quarter from upper-left to center, and one quarter from bottom-right to center. All orders yield a reward of 2 except those that go from right to center, which yield a reward of 4. For this domain, the best policy first assigns drivers to travel to the bottom-right region, and once they are there, assign them to the bonus reward trips back from right to center. A policy that minimizes pickup distance will fail to value trips to the bottom-right more than trips to top-left, and therefore yield suboptimal behavior. On the other hand, a policy which is greedy with respect to rewards will always select the bonus order regardless of driver location. In effect the policy "skips" the price 2 order that will ferry a driver out to the bottom-right region, and is therefore also suboptimal. An illustration of the surge domain can be found in the appendix.

Hot/Cold domain

In the Surge domain, the advantage of traveling to the bottom right region is clear; it is directly tied to the price of orders found in that region (4 vs. 2). In the Hot/Cold domain, the agent must learn a more subtle advantage. Order pickup locations are located uniformly along the top edge of the unit square, called the "hot region". Half of the orders end uniformly along the bottom edge of the unit square, called the "cold region" and half end uniformly in the hot region. Order price is given by the *Euclidean distance from order pickup to order dropoff locations*. The hot region can be thought of as a busy area of downtown, while the cold region represents surrounding suburbs. Despite orders to the cold region having higher price (since they are longer), it is generally more advantageous for drivers to stay in the hot region, since they can quickly pick up new orders. In other words, the advantage is entirely *temporal*. An illustration of the Hot/Cold domain can be found in the appendix.

We compare SD-DQN and MD-DQN with 3 other algorithms: myopic revenue maximization (MRM), myopic pickup distance minimization (MPDM), and local policy improvement (LPI). MRM always assigns the highest value order to an available driver. MPDM always assigns the closest order to an available driver. LPI [**?**] discretizes the environment into a 20x20x144 spatiotemporal grid and performs tabular TD(0). We report average revenue, pickup distance, and served percentages with error bars over 100 episodes. Each episode lasts 5000 time units, which allows each driver to serve approximately 1000 orders.

Results can be found in **??** and **??**. There are a few key takeaways from these results. First, all learning methods, including LPI, outperform myopic strategies across the board. SD-DQN and MD-DQN also typically improve over LPI, though the margin is considerably smaller. Finally, it is not clear what situations favor SD-DQN vs. MD-DQN. For instance, one might expect SD-DQN to do comparatively better in high demand situations, where there would seem to be a reduced need for coordination, however this is not the case.

	Low Demand			High Demand		
Algorithm	Revenue	Pickup distance	Served %	Revenue	Pickup distance	Served %
MRM	29447	.33	73.6	35939	.54	18.1
MPDM	32279	.178	86.3	42842	.016	34.2
LPI	31112	.245	80.0	50147	.046	33.6
SD-DQN	31860 ± 448	.279 ± .0005	78.2 ± .19	50323 ± 87	.045 ± .02	33.54 ± .09
MD-DQN	33700 ± 225	.177 ± .0001	88.23 ± .28	49031 ± 130	.056 ± .0012	32.79 ± .05

Table 1.2: Surge Domain

	Low Demand			High Demand		
Algorithm	Revenue	Pickup distance	Served %	Revenue	Pickup distance	Served %
MRM	50953	1.04	31.5	52094	1.11	8.7
MPDM	56546	.535	53.8	58287	.508	16.5
LPI	58173	.45	60.64	76840	.1545	30.06
SD-DQN	58580 ± 124	.4609 ± .007	59.26 ± .13	78552 ± 212	.1108 ± .003	39.25 ± .04
MD-DQN	58893 ± 181	.5158 ± .008	52.97 ± .027	78860 ± 285	.111 ± .012	33.625 ± 1.16

Table 1.3: Hot/Cold Domain

Illustrative Domains with Repositioning

Whereas the previous experiments only deal with dispatching, we now examine our methods on domains where drivers can both be dispatched and repositioned.

Repositioning Hot/Cold Domain

The first such environment is the same as the previous Hot/Cold domain, except we impose a *broadcasting radius* d_{bcast} on drivers. This means that drivers may only pair with orders if they are within d_{bcast} units of the driver. Otherwise, the driver may only take a repositioning action. For this domain we set $d_{bcast} = 0.3$ If a driver matches to an order that ends in the cold region, the agent must learn to reposition that driver from the cold region towards the hot region (which consists of approximately 4 consecutive "move up" repositioning actions)

	Low Demand			High Demand		
Algorithm	Revenue	Pickup distance	Served %	Revenue	Pickup distance	Served %
MRM-random	932	.199	4.2	911	.177	1.8
MPDM-random	939	.174	8.1	936	.161	2.5
MRM-demand	4861	.180	34.1	4902	.178	8.1
MPDM-demand	5234	.1624	53.2	5644	.164	15.9
SD-DQN	5478 ± 188	.1615 ± .03	57.5 ± .31	7387 ± 41	.0781 ± .008	33.8 ± .43
MD-DQN	5717 ± 213	.1879 ± .05	54.5 ± .25	7309 ± 56	.1519 ± .04	24.2 ± .22

Table 1.4: Hot/Cold with repositioning

so the driver can pair with additional orders. As with the dispatch-only experiments, we present results for high and low demand regimes.

We compare our methods against two versions of MPDM with repositioning: MPDM-random and MPDM-demand. If a driver has no orders within broadcast distance, MPDM-random *randomly* selects a repositioning action, whereas MPDM-demand repositions the driver towards the nearest order. As we can see in **??**, SD-DQN and MD-DQN both maintain their advantage over baselines when required to learn repositioning and dispatching together.

Distribute Domain

While Hot/Cold with repositioning tests an important aspect of learning - namely, the ability of MD- and SD-DQN agents to reposition drivers to locations where they can pick up new orders, this repositioning behavior is quite simple in that it is *uniform across drivers*. This means that the agent can always reposition drivers in the same manner (i.e. "if in cold region, go to hot region"). In order to test whether our methods can learn non-uniform repositioning behavior, we introduce a class of “Distribution environments” where drivers must be repositioned so as to match their spatial distribution with a fixed future order distribution. A Distribute domain operates in two phases. In the first phase, the environment resets with k drivers and no orders in the system, so drivers may only reposition during this phase. In the second phase, k orders appear according to a fixed spatial distribution, and drivers can match to orders if they are within a given broadcast radius d_{bcast}. The second phase only lasts long enough to allow drivers to reposition one more time before all orders cancel and the environment is reset. We alter the reward function so that each order matching action receives +1 reward. Order destinations are designed to be far away from start locations so that each driver may only serve one order per episode. As a result, the episodic return is proportional to the number of orders served, so we may interpret the episode score as a measure of how well the agent arranges driver supply in phase 1 with order demand in phase 2.

In our experiments, the distribution of orders always consists of two small patches in the top left and bottom right parts of the unit square. The order start locations are sampled uniformly within each patch. The total number of orders in each patch is fixed across episodes, and we denote it fractionally. An even order split between patches (eg 10 orders in both patches) is denoted 50/50. If 80 percent of orders are in the first patch and 20 percent are in the second patch, we denote it as 80/20. Visualizations of the Distribute domain are in the appendix.

Results for 20-driver distribute domains are shown in **??**. We include the optimal served percentage (which is 100 %) and the "uniform optimal" served percentage. This quantity reflects the maximum score one can obtain if the repositioning behavior is uniform across drivers. SD- and MD-DQN are able to get near optimal test scores when the demand is balanced. However, in the 80/20 task, only SD-DQN was able to escape the uniform optimum. For all experiments it was critical to allow for sustained high exploration. Specifically, in all experiments we used an ϵ-greedy behavior policy where ϵ was linearly annealed epsilon from 1.0 to 0.2 over the first 1000 episodes. Test performance is averaged over 10 episodes. Also, we ran each experiment 4 times changing only the random seeds. We found that final performance across seeds was nearly identical in all experiments. Learning curves can be found in the appendix.

We also used the distribute domain to test the saliency of global state information in the learning process of SD-DQN. Traditionally, independent learning in games assumes that agents only have a partial view of state at decision time [**?**]. In contrast, SD-DQN receives full state information as input. We demonstrate the salience of global state through a small distribute domain in which there are 4 drivers and a 75/25 split i.e., 3 orders appear in

Algorithm	50/50 Served %	80/20 Served %
Optimal	100%	100%
Uniform Optimal	50%	80%
SD-DQN	96 ± .13%	92 ± .72%
MD-DQN	95 ± .11%	80 ± 3.42%

Table 1.5: Distribute Domain with 20 Drivers

one region and 1 order appears in the other. We then trained SD-DQN with and without the inclusion of global context. Without global context, the network becomes stuck in the uniform optimal strategy that sends all drivers to the 3 order region. A graph comparing served percentage with and without global state can be found in the appendix.

Non-repositioning Historical-Statistics Real-World Domain

Finally, we test our method in more realistic dispatching environments. We refer to the first of these as the Historical-Statistics domain, because it derives distributional order and driver generation parameters from historical data. This first realistic simulator *does not include repositioning*. Specifically, we used 30 days of dispatching data from DiDi Chuxing's GAIA dataset [**?**], which contains spatial and temporal information for tens of millions of trips in one of the largest cities in China. To build the simulator from this data, we first covered the city in a square 20 by 20 grid, and extracted Poisson parameters $\kappa_{x,y,t}$ where x is an order start tile, y is an order end tile, and t is the time of day in hours. This results in $400 \times 400 \times 24 = 3.84$ million parameters which we use specify an order generation process. In addition to these, we also extract the average *ETA*, as well as its variance, for each (x, y, t) triple. When a driver is assigned to an order, the simulator computes a Gaussian sample (using the historical mean and variance) of the ETA t_1 from the driver's position to the order's start location, and another sample of the ETA t_2 for the order's start location to the order's end location. The driver will become available at the

	.1% scale			1% scale		
Algorithm	Revenue	Pickup ETA	Served %	Revenue	Pickup ETA	Served %
MRM	10707	22.74	20.9	117621	22.32	20.16
MPDM	11477	11.99	31.6	134454	6.1	36.79
SD-DQN	12085 ± 19	19.15 ± .16	24.96 ± .11	146182 ± 244	15.07 ± .11	27.64 ± .09
MD-DQN	11145 ± 78	21.77 ± .62	21.38 ± .32	122671 ± 698	19.50 ± .52	22.14 ± .76

Table 1.6: .1% and 1% scaled real data

Algorithm	Revenue	Pickup ETA	Served %
MRM	1112340	22.37	20.04
MPDM	1333180	6.2	29.4
SD-DQN	1391141	17.28	25.3
MD-DQN	1161780	20.05	23.17

Table 1.7: 10% scaled real data

order's end location in time $t_1 + t_2$. Orders price is equal to $max(5, t_2)$, where t_2 is given in minutes. Driver entry and exit parameters are also derived from data. For each tile-hour triple (x, y, t) we compute the poisson parameter from driver arrival, and the duration that the driver remains in the system is given by a poisson parameter that is a function only of t.

To control computational costs we control the *scale* of the MDVDRP via a scaling parameter $0 < \lambda \leq 1$. All order and driver generation parameters are multiplied by λ. For example, a 1% scaled environment means that we multiplied all generation parameters by 0.01. We present results for 3 scale regimes: .1%, 1% (**??**), and 10% (**??**). For .1% and 1% we report values with standard errors across 100 episodes, and for 10% we report values with standard error across 10 episodes.

Across all scales, SD-DQN outperforms both myopic baselines, while MD-DQN generally only performs slightly above myopic revenue maximization.

Algorithm	Revenue	Pickup ETA	Served %
MRM	414	2.60	44
MPDM	511	1.1	79
MPDM-random	494	.9	73
MPDM-demand	502	.8	76
SD-DQN	542	1.2	75
MD-DQN	474	1.8	53

Table 1.8: 10% region real data

Repositioning Historical-Order Real-World Domain

We also experiment with a simulator that uses historical days of orders instead of generating orders randomly from historical statistics. The GAIA dataset provides 30 days of orders in the city of Chengdu. We first found a small spatial region of Chengdu for which 10% of orders both start and end in that region. This region defined the historical data simulator. We then created 30 *order generation schemes*. Precisely, when the environment is reset, it randomly selects one of the 30 days, and generates orders exactly according to how orders arrived on that day. We used a fixed number of drivers (100), and a fixed speed (40 km/h). An illustration of this environment can be found in the appendix. For SD-DQN and MD-DQN, we impose a 2 kilometre broadcast radius. We compare performance against the standard non-repositioning baselines of myopic revenue maximization (MRM) and myopic pickup distance minimization (MPDM), both of which have no broadcast distance and no repositioning. We also compare against MPDM-random and MPDM-demand. The broadcast distance used in these repositioning baselines is 2 kilometers.

We obtain the similar relative performance as the historical statistics domain, with MD-DQN performance above MRM but below MPDM, and SD-DQN outperforming all myopic strategies.

1.0.6 Conclusion

We performed a detailed empirical study of two reinforcement learning approaches to multi-driver vehicle dispatching and repositioning problems: single-driver Q-learning and multi-driver Q-learning. Both approaches leverage a global representation of state processed using attention mechanisms, but differ in the form of Q-learning update used. We found that, while one can construct environments where MD-DQN is superior, typically SD-DQN is competitive. Furthermore we applied these methods to domains built from real dispatching data, and found that SD-DQN is able to consistently beat myopic strategies across scales, as well as with and without repositioning actions.

conclusion goes here

1.0.7 Environment descriptions and visualizations

The following figures show and describe MDVDRPs in general, as well as particular visualizations for all dispatching environments.

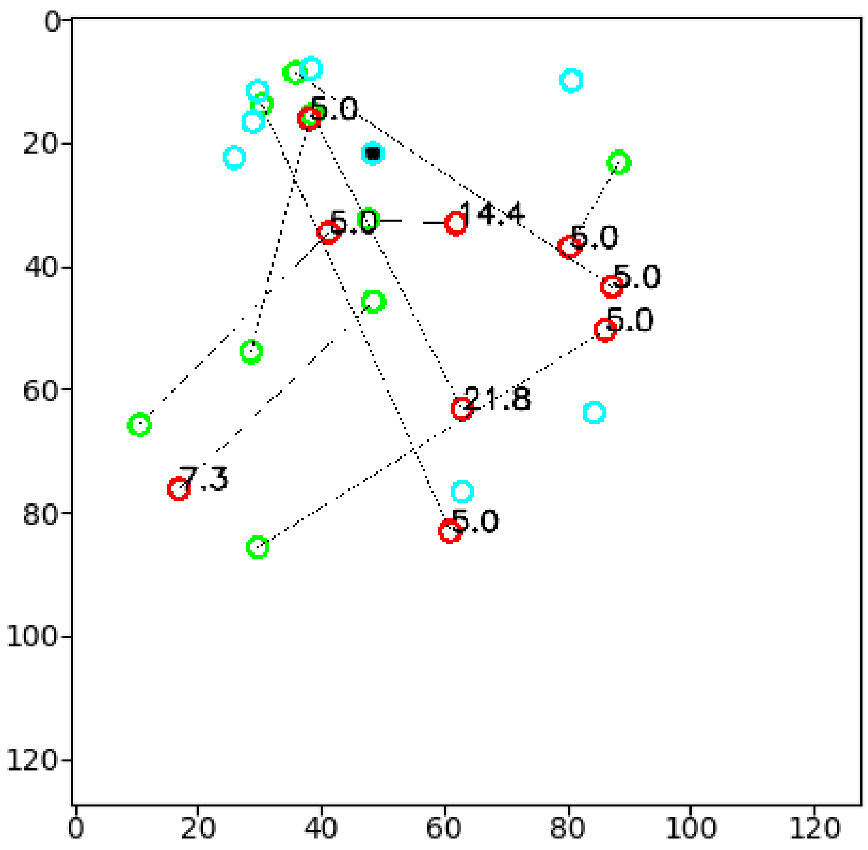

Figure 1.2: A visual representation of a dispatching state. Blue dots represent drivers. The black centered driver located at position (60, 20) is available, and all others are dispatched. Orders start at red dots and end at the connected green dot. Order prices are denoted above their pickup location

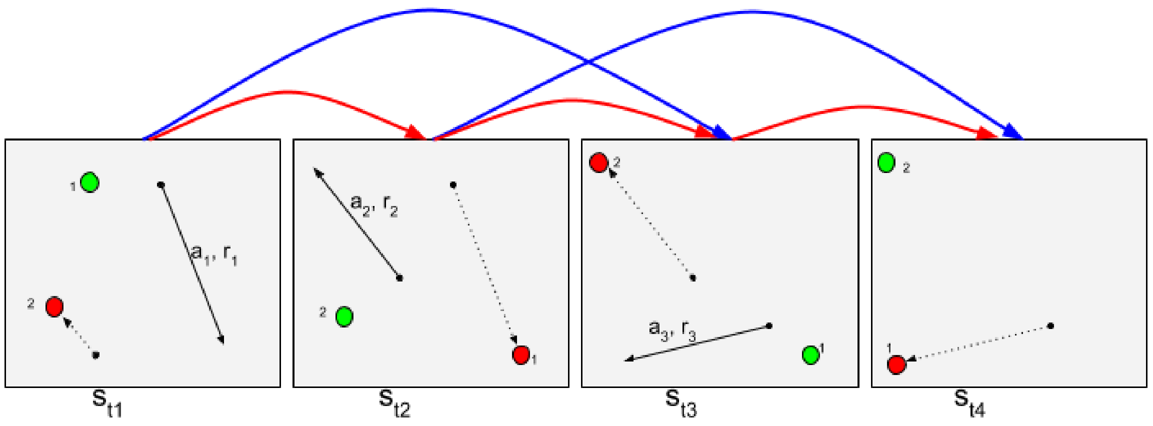

Figure 1.3: The above image shows a length 4 trajectory. The currently available driver is green, dispatched driver is red, and the order that the available driver accepts at time t_i is a_i and has price r_i. The accepted order at time t_i is labeled by its action name and price, (a_i, r_i) and travels from the solid black dot to the terminal arrow. SD-DQN transitions are indicated by blue arrows above state, e.g. transition $(s_{t1}, a_1, r_1, s_{t3})$, which is driver-centric with respect to driver 1. MD-DQN transitions are indicated by red arrows e.g. transition$(s_{t1}, a_1, r_1, s_{t2})$, which transitions from a state where driver 1 is available to a state where driver 2 is available.

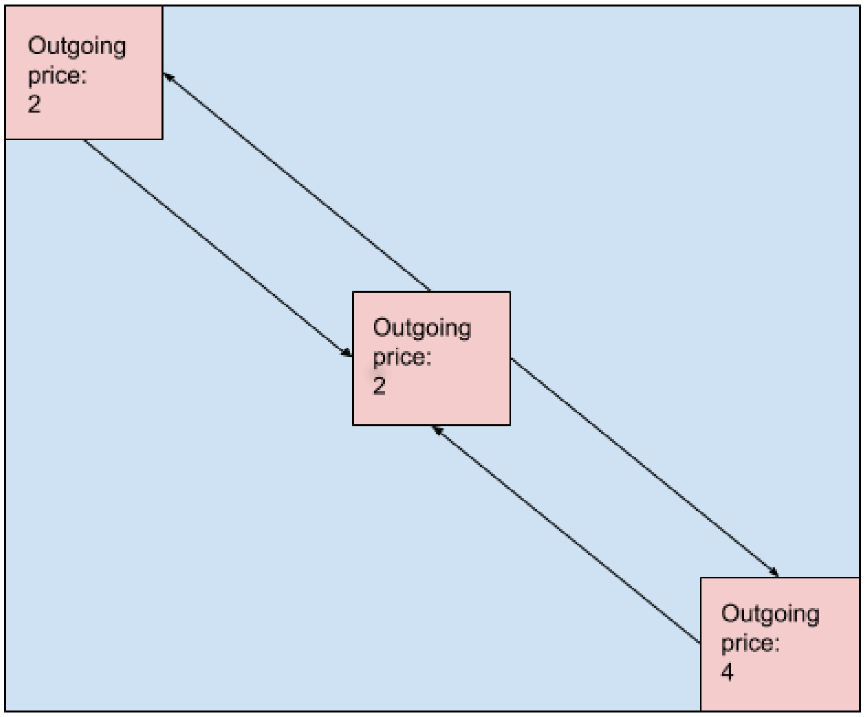

Figure 1.4: Surge domain. Orders travel between the three red squares. Each square is labeled with its outgoing order value. Within each square, order start and end locations are generated uniformly randomly.

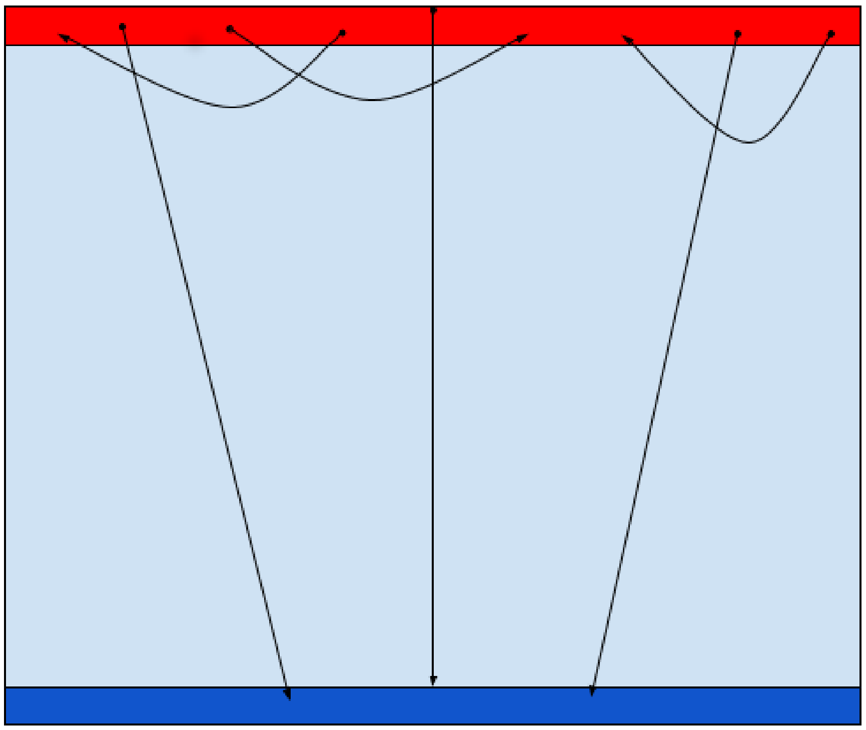

Figure 1.5: Hot/Cold domain. Orders all begin in the red bar, with their positions generated uniformly randomly. For the destination, a fair coin is flipped to decide whether the order ends in hot or cold, and then the exact position is sampled uniformly randomly in the designated region.

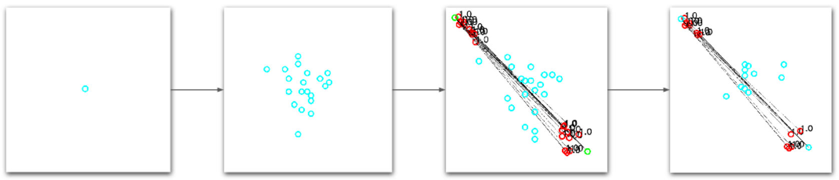

Figure 1.6: Distribute domain. Drivers begin in the center of the region. They then proceed with 5 steps of repositioning. At the 6th timestep, orders appear in the two corners. Drivers that are within .3 units of an order start, denoted by a red circle, are assigned. All orders end in the opposing cornerâĂŹs green dot so that trips are long enough that a single driver can only satisfy one order per episode. After two timesteps, all remaining orders cancel and the environment resets.

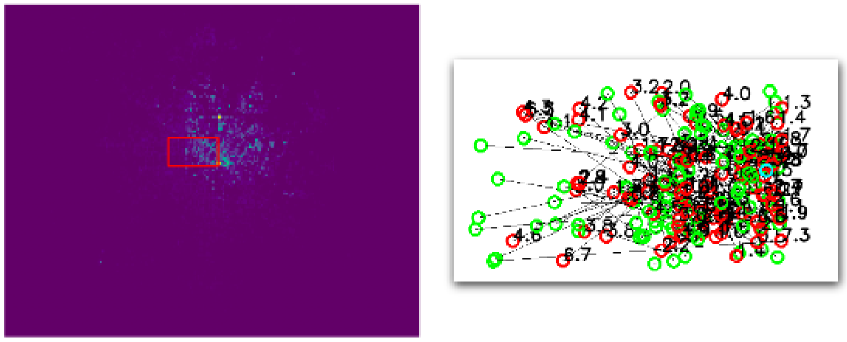

Figure 1.7: Historic data order start distribution and corresponding simulator rendering. The red box indicates the spatial region that is selected for the simulator. an average of 10 percent of orders start and end in this region, which roughly correspond to an edge of downtown and some outlying areas.

1.0.8 Graphs

In the following graphs, the units of the x-axis are in *episodes*. For the nonrepositioning illustrative domains, an episode lasts 5000 time units. In the Repositioning Hot/Cold domain an episode is 500 time units. The distribute domain lasts 7 time units. The realistic domains last for 1440 time units in the realistic simulator. Graphs for Distribute domains show served percentage on the y-axis, while all other curves are environment reward.

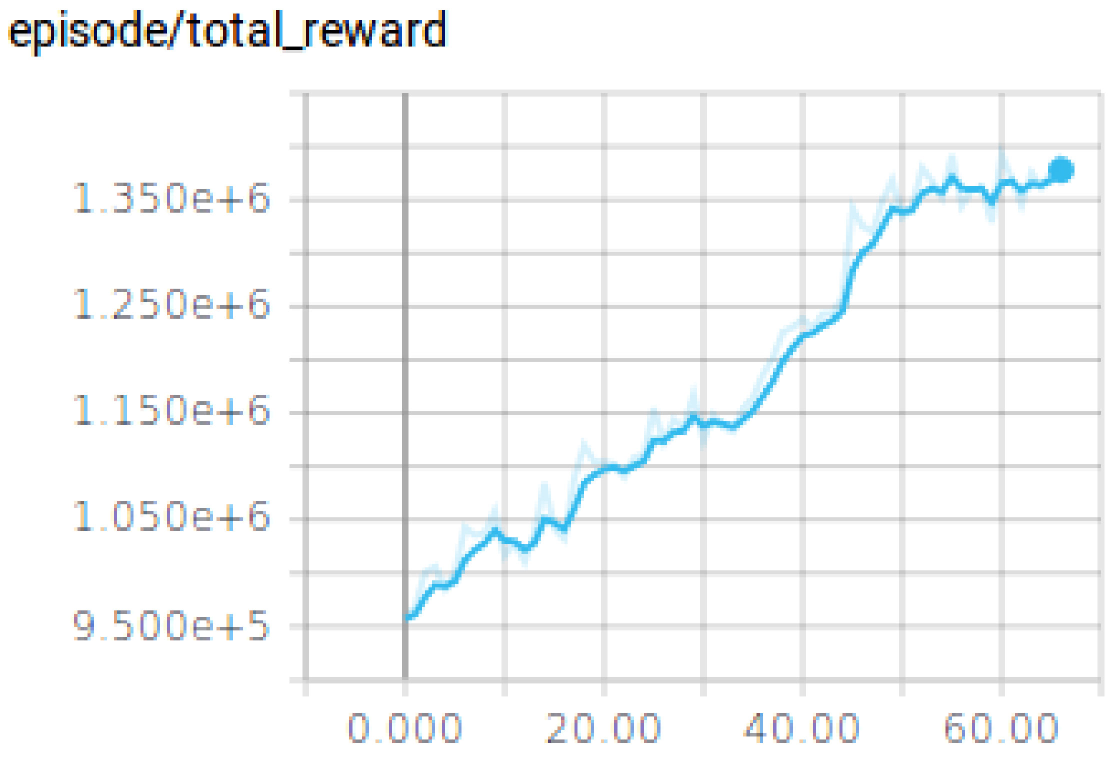

Figure 1.8: SD-DQN on 10% historical statistics simulator

1.0.9 Distribute Domains

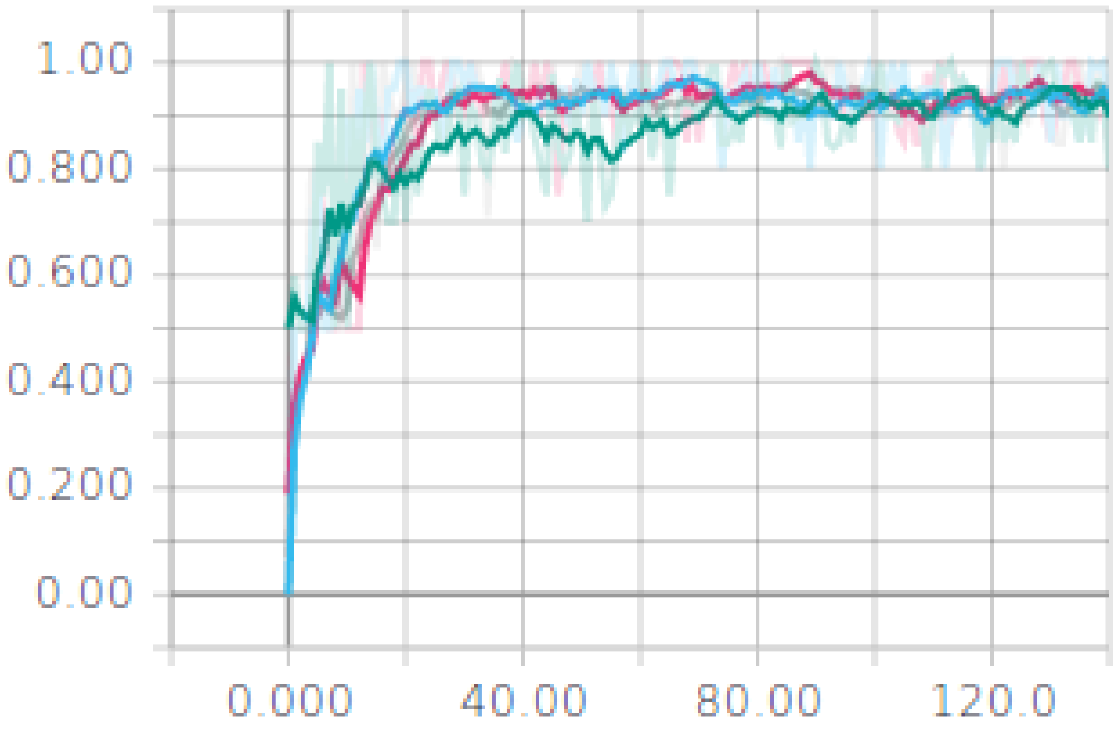

Figure 1.9: SDDQN served percentage on 20 driver 50/50 distribute domain

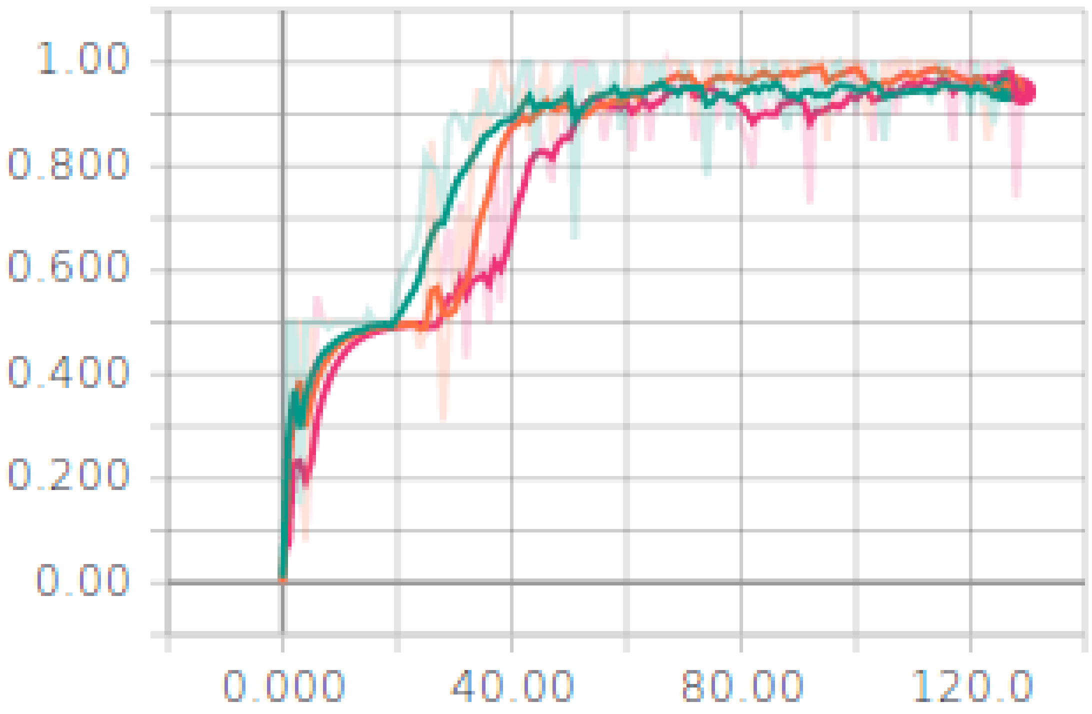

Figure 1.10: MDDQN served percentage on 20 driver 50/50 distribute domain

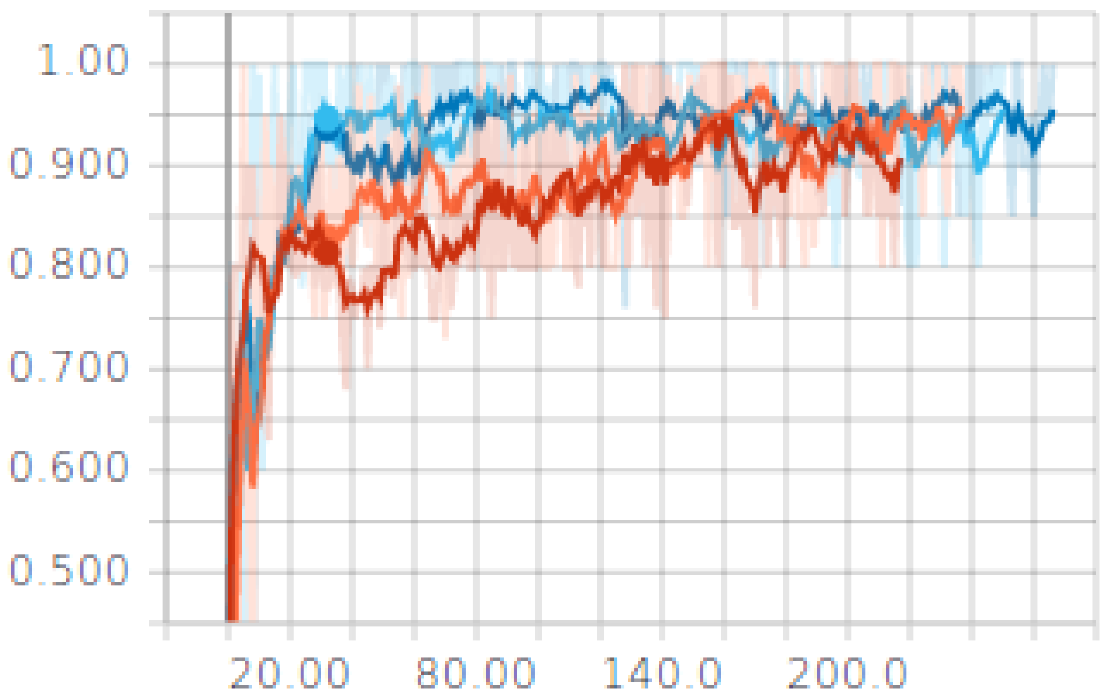

Figure 1.11: SDDQN served percentage on 20 driver 80/20 distribute domain

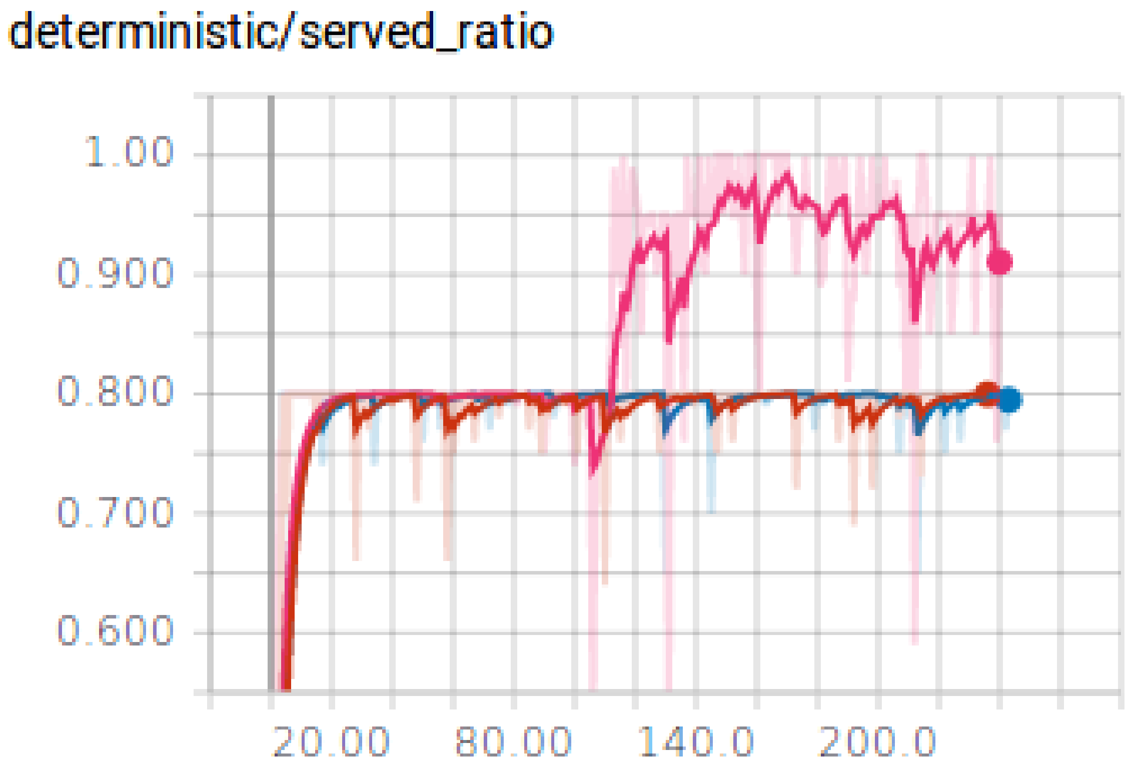

Figure 1.12: MDDQN served percentage on 20 driver 80/20 distribute domain

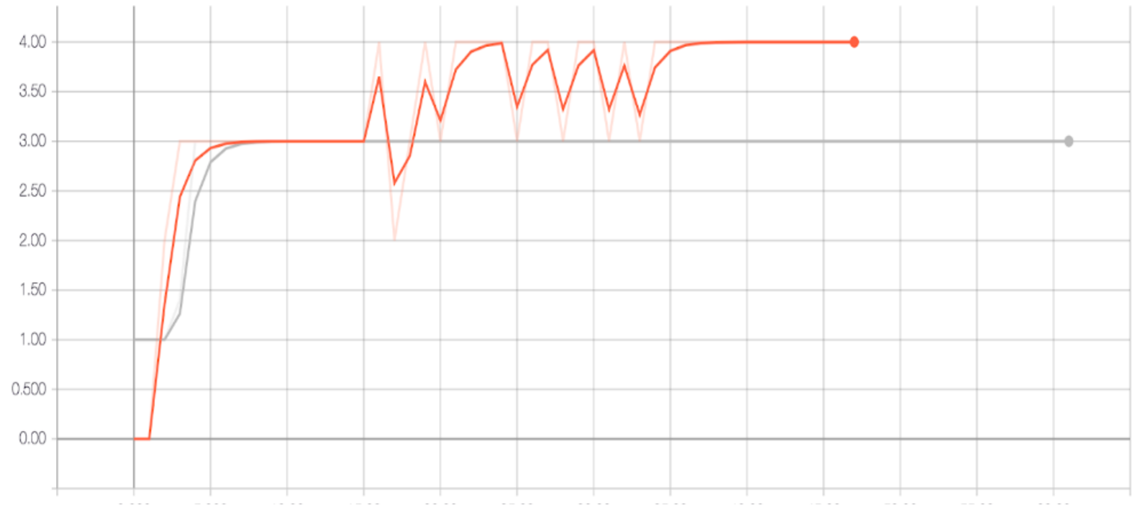

Figure 1.13: SDDQN served percentage on 4 drivers 75/25 distribute domain. The orange curve shows SD-DQN performance when global context is included during the Q-value querying while the gray curve does not include global context. SDDQN without global context learns a policy that is uniform across drivers, and so it never escapes the uniform optimum.

1.0.10 Historical Data Domain

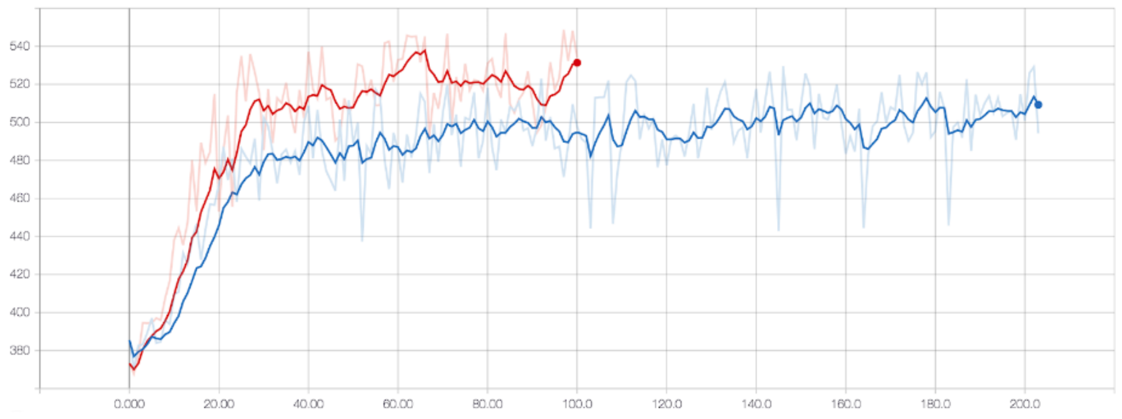

Figure 1.14: SD-DQN revenue on 10% historical simulator. The blue curve shows learning when revenue itself is used as the reward function. The red curve shows revenue when negative pickup distance is used as the reward function.

1.0.11 Training Details

20 Driver Static Assignment Problem

ϵ-exploration is annealed linearly over $10,000$ episodes from 1.0 to 0.03, and the target network is updated every 500 training steps. The replay buffer maintains the 10000 most recent transitions, and is initialized with 5000 samples. We used a learning rate of 0.001. Unlike the following domains, the reward function is given as the negative Euclidean distance between the selected driver-order pair.

All other illustrative domains

For SD-DQN we used a size 20000 replay buffer, learning rate 0.001, and we annealed ϵ-exploration linearly over 100 episodes. We update the target network every 500 training updates, and perform a training update every environment step. For MD-DQN we used a size 20000 20-step Q-learning, a learning rate of 0.0001, and annealed ϵ-exploration linearly over 100 episodes. We found it critical to stability to reduce the learning rate and perform *n*-step Q-learning in MD-DQN.

Chapter 2

Stochastic Potential Games

2.1 Introduction

In its early years, game theory focused on the static concepts of equilibria in normal-form games, proving their existence, and computing their behavior. From the period of the 1980's until the early 2000's there was a shift in research focus towards *learning* equilibria. This was motivated both by descriptive applications in economics as well as prescriptive applications in engineering. This led to a more in depth study of simple learning rules such as best response, fictitious play, log-linear learning, and joint strategy fictitious play. A number of researchers obtained convergence results for special subclasses of games Two games of particular interest are zero-sum games [Shapley 1953] and potential games [Shapley 1996]. Both classes of games admit pure Nash equilibria, and futhermore a number of the simple learning rules listed above converge to Nash equilibria in these types of games.

While a great deal of work has been done on learning to play equilibria strategies in normal-form games, the same cannot be said for stochastic games. The defining feature of stochastic games is that they introduce state dynamics, with each state admiting a normal-form game. Stochastic games are of interest both from a theoretical and practical point of view. They are the natural generalization of Markov decision processes (MDPs) to the multiagent setting. As such they extend the modelling capacity of MDPs, and can be used to model markets, bargaining, and routing problems to name a few.

This work aims to study convergence properties of classes of stochastic games. As such, we extend the definition of normal-form potential games to the stochastic game setting. We then study the existence and properties of their Nash equlibria, and consider convergent learning dynamics.

2.2 Background

We begin with a review of normal form games, normal-form potential games, and log-linear learning in normal form potential games. Then we will review basic concepts of stochastic games and their equilibiria.

2.2.1 Normal-form games

Definition 2.2.1. An n-player normal-form game $\Gamma = (\mathcal{A}^i, u^i)$ consists of an action set $\mathcal{A}^i$ for each player i and a collection of *utility functions*

$$u^i : \times_{j\in\{1,\ldots,n\}}\mathcal{A}^j \to \mathbb{R}$$

In this work the $\mathcal{A}^i$ will always be finite sets.

Let $\Delta\mathcal{A}^i$ denote the simplex of discrete probability distributions over $\mathcal{A}^i$. We will refer to an element x^i of this distribution as a player strategy, and a *pure* strategy if the support of x^i is a single action a^i. By a small abuse of notation we will call such pure strategies by their action name a^i. The utility functions can be linearly extended to the domain $\times_{j\in\{1,\ldots,n\}}\mathcal{A}^j$, and we will generally think of the u^i's as having this enlarged domain.

A list of stragies, one per player: $x = (x^1, \ldots, x^n)$ is called a *joint strategy*. Often, it is useful to consider two joint actions that differ by one or two players' actions. For notational convenience, we define the joint action $(x^1, \ldots, x^{i-1}, y^i, x^{i+1}, \ldots, x^n)$ by $x\backslash y^i$ and the joint action $(x^1, \ldots, x^{i-1}, y^i, x^{i+1}, \ldots, x^{j-1}, y^j, x^{j+1}, \ldots, x^n)$ by $x\backslash y^i y^j$. A joint pure strategy is one where all players' strategies are pure.

Let $G = (\mathcal{A}^i, u^i)$ and $H = (\mathcal{A}^i, v^i)$ be two normal-form games with the same action sets. We can define sum of these two games as the game $G + H = (\mathcal{A}^i, u^i + v^i)$. The game

$G - H$ can be defined similarly.

Definition 2.2.2. For a game $\Gamma = (\mathcal{A}^i, u^i)$ and player i, a strategy $x^i \in \Delta\mathcal{A}^i$ is said to be a best response to $\{x^j\}_{j \neq i}$ if

$$u^i(x^1, \ldots, x^i, \ldots, x^n) \geq u^i(x^1, \ldots, y^i, \ldots x^n)$$

for any $y^i \in \mathcal{A}^i$.

A Nash equilibrium is simply a joint strategy such that each strategy is a best response to all others:

Definition 2.2.3. A Nash equilibrium is a set of strategies $x^i \in \Delta\mathcal{A}^i$ such that for all i, the player strategy x^i is a best response to $\{x^j\}_{j \neq i}$

A *pure* Nash equilibrium is a Nash equilibrium such that each player strategy is a pure strategy.

2.2.2 Normal-form potential games

Potential games are a class of normal-form games that admit pure Nash equilibria as well as convergent learning dynamics.

Definition 2.2.4. A potential game is an n-player game $\Gamma = (\mathcal{A}^i, u^i)$ together with a function (called a *potential function*)

$$\varphi : \times_i \mathcal{A}^i \to \mathbb{R}$$

such that for any joint strategy $x = (x^1, \ldots, x^n)$ and player strategy $y^i \in \mathcal{A}^i$,

$$\varphi(x \backslash y^i) - \varphi(x) = u^i(x \backslash y^i) - u^i(x)$$

This equation says that when a player unilaterally changes the joint action (that is, they change their action and all other players maintain the same action), the change in that players utility is equal to the change in the potential function. This also means that potential functions are unique up to constant shifts.

The following are several useful facts about potential functions and their equilibria.

Theorem 2.2.5. *Let* Γ *be a potential game with potential function* φ*. Any pure joint strategy* x *that maximizes* φ *is a Nash equilibria.*

Theorem 2.2.6. *Let* $\Gamma = (\mathcal{A}^i, u^i)$ *be a normal-form game. Then* Γ *is a potential game if and only if for any pure joint stragegy* a*, players* i *and* j*, and pure player strategies* $b^i \in \mathcal{A}^i$ *and* $b^j \in \mathcal{A}^j$

$$u^i(a \backslash b^i b^j) - u^i(a \backslash b^j) + u^j(a \backslash b^j) - u^j(a) = u^j(a \backslash b^i b^j) - u^j(a \backslash b^i) + u^i(a \backslash b^i) - u^i(a)$$

2.2.3 Log-linear learning

2.2.4 stochastic games

Finally, we introduce the definition of a stochastic game and their Nash equilibria.

Definition 2.2.7. An n-player stochastic game Γ consists of a state space $\mathcal{S}$, action sects $\mathcal{A}^i$ for each player, and games $\{G_s\}_{s \in \mathcal{S}}$ which will be called "stage games", and trainsition probabilities $P_{ss'}(x)$ for each state pair $s, s' \in \mathcal{S}$ and joint strategy x.

In this work we will always assume that $|\mathcal{S}|$ is finite and that the action sets are finite.

The stochastic game framework lies at the intersection of game theory and control theory. By introducing a state space and transition dynamics it extends the normal-form game from game theory. A normal-form game can be viewed as a stochastic game with one state and trivial transition dynamics. On the other hand, it extends the Markov decision process (MDP) framework to the multi-agent setting. An MDP can be viewed as a stochastic game with one player.

It will also be useful to restrict the form of stochastic games for later analysis in the paper. For this purpose we introduce the following definition:

Definition 2.2.8. A stochastic game Γ is said to be *layered* if there exists a partition of the state space $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_T$ such that for $s_i \in \mathcal{S}_i$ and $s_j \in \mathcal{S}_j$ we have

$$P_{s_i s_j} = 0$$

if $j \neq i+1$.

In normal-form games, each player chooses a player strategy. In stochastic games, a player must choose a strategy in every state. We refer to such a choice as a behavior, and denote it by $\pi^i \in \times_{s\in\mathcal{S}}\mathcal{A}^i$. The specific strategy employed in state s will be denoted by π^i_s. A joint behavior is a collection of behaviors for all players $\pi = (\pi^i, \dots, \pi^n)$. It is useful to introduce notation for joint behaviors that differ by one or two players' behaviors. Let τ^i be a behavior for player i. We denote the joint behavior $(\pi^1, \dots, \pi^{i-1}, \tau^i, \pi^{i+1}, \dots \pi^n)$ by $\pi\backslash\tau^i$. Similarly, if we change two players actions, we denote the joint strategy by $\pi\backslash\tau^i\tau^j$. Finally, we will often modify a joint behavior π be changing a players strategy in a single state s from pure strategy a^i_s to b^i_s. We will denote the modified joint behavior by $\pi\backslash b^i_s$ or $\pi\backslash b^i$ when the particular state is clear.

In normal-form games, players wish to maximize their utilities. In stochastic games, players wish to maximize their returns. A return is simply the sum of utilities from the stage games they encounter. In analogy with the reinforcement learning literature, we let $V^i_\pi(s)$ denote the expected return for player i starting from state s when players use the joint behavior policy π. We let $\mathbb{Q}^i_\pi(s, a)$ denote the expected return for player i starting from state s when they first take action a in state s while other players play π_s, and then all players play π in subsequent states.

In [Leslie] the authors study the convergence properties of best response dynamics in *zero sum* stochastic games. For this purpose they consider the following "continuation games". Consider a stochastic game Γ and a set of "continuation vectors" $z^i = \{z^i_s\}_{s\in S}$, one for each player i. We refer to the entire collection as z. Let $G_s(z)$ be a normal-form game with payoffs

$$u^i_s(a) + \sum_{s'\in\mathcal{S}} P_{ss'}(a) z^i_{s'}$$

In order to define convergent dynamics in zero sum games, the authors make use of an iterative process that alternates between solving a (zero-sum) continuation game and using the solution to define new continuation payoffs with creates a new continuation to solve. Critical to their analysis is that at each step, their continuation game is itself a zero sum game. Motivated by this analysis, we aim to define stochastic potential games in such a way that every continuation game is a potential game.

2.2.5 Stochastic Potential Games

Definition 2.2.9. We say that a stochastic game Γ has *modular dynamics* if, for any two players i and j, any joint action $a \in \times_i \mathcal{A}^i$, and any states $s, s' \in \mathcal{S}$:

$$P_{ss'}(a) + P_{ss'}(a \backslash b^i b^j) = P_{ss'}(a \backslash b^i) + P_{ss'}(a \backslash b^j)$$

Definition 2.2.10. We say that a stochastic game Γ is a stochastic potential game if

1. Every stage game G_s is a potential game and
2. Γ has modular dynamics

Lemma 2.2.11. *Let G and H be two potential games with potential functions φ and ψ respectively. The games $G + H$ and $G - H$ are potential games with potential functions $\varphi + \psi$ and $\varphi - \psi$ respectively.*

Theorem 2.2.12. *Consider a stochastic game* Γ. *Every continuation game is a potential game if and only if* Γ *is a stochastic potential game.*

Proof. Assume Γ is a stochastic potential game. Fix z a set of continuation vectors for each player and consider the continuation game $G_s(z)$ with payoffs

$$u_s^i(a) + \sum_{s' in \mathcal{S}} P_{ss'}(z_{s'}^i)$$

G_s is a potential game since Γ is a stochastic potential game, and so by the lemma it suffices to show that the game $D_s(z) = G_s(z) - G_s$ is a potential game. This game has utilities

$$\sum_{s' \in \mathcal{S}} P_{ss'}(a) z_{s'}^i$$

We will refer to $D_s(z)$ as the "delay game".

By [theorem from background], it suffices to check that for any players i and j, any joint action a, and any player actions $b^i \in \mathcal{A}^i$ and $b^j \in \mathcal{A}^j$ we have

$$\sum_{s'} P_{ss'}(a\backslash b^i b^j) z^i_{s'} - \sum_{s'} P_{ss'}(a\backslash b^j) z^i_{s'} + \sum_{s'} P_{ss'}(a\backslash b^j) z^j_{s'} - \sum_{s'} P_{ss'}(a) z^j_{s'} =$$
$$\sum_{s'} P_{ss'}(a\backslash b^i b^j) z^i_{s'} - \sum_{s'} P_{ss'}(a\backslash b^i) z^j_{s'} + \sum_{s'} P_{ss'}(a\backslash b^i) z^i_{s'} - \sum_{s'} P_{ss'}(a) z^i_{s'}$$

which can be rearranged to

$$\sum_{s'} \left[P_{ss'}(a) + P_{ss'}(a\backslash b^i b^j) - P_{ss'}(a\backslash b^i) - P_{ss'}(a\backslash b^j)\right] z^i_{s'} =$$
$$\sum_{s'} \left[P_{ss'}(a) + P_{ss'}(a\backslash b^i b^j) - P_{ss'}(a\backslash b^i) - P_{ss'}(a\backslash b^j)\right] z^j_{s'}$$

And, since, the dynamics are modular, all terms in both the left-hand and right-hand sum are zero. Therefore the continuation game is a potential game.

Now, assume every continuation game is a potential game. Then in particular this is true when z is identically zero, and therefore the stage games of Γ are potential games. Finally we need to show that this implies Γ has modular dynamics.

Fix a players i and j, states s and s', joint strategy a and player strategies b^i and b^j. Let z^j be the zero vector, and z^i be zero everywhere except at state s'. Then equation [reference equation above] reduces to

$$P_{ss'}(a) + P_{ss'}(a\backslash b^i b^j) - P_{ss'}(a\backslash b^i) - P_{ss'}(a\backslash b^j) = 0$$

Hence Γ has modular dynamics.

$\square$

2.2.6 Stochastic Global Potential Games

Notice that while our definition of SPGs places a potential at each state, there is not necessarily a unified potential function defined over the stochastic game. A different approach to extending potential games is to note that the original definition involves a potential function over strategies, and so the stochastic version should define a potential function over *behaviors*. We will call this a *stochastic global potential game* (SGPG).

Definition 2.2.13. A stochastic global potential game is a stochastic game Γ together with a *global potential function*:

$$\Phi : \mathcal{S} \times \Pi^1 \times \cdots \Pi^n \to \mathbb{R}$$

such that for every state s, joint behavior π, and player i with behavior τ^i,

$$\Phi(\pi \backslash \tau^i) - \Phi(\pi) = V^i_{\pi\backslash\tau^i}(s) - V^i_\pi(s)$$

Stochastic global potential games are amenable to reinforcement learning methods because they admit acyclic "strict better reply graphs". This essentially means that if players sequentially make improvements to their own behaviors, their joint behavior will eventually constitute a Nash equilibrium. However, determining whether a stochastic game is an SGPG is not straightforward. A priori, there is no way to check whether a global potential function exists other than to construct one, and the temporally extended nature of the global potential function may make this difficult. In contrast, SPGs are characterized by temporally local conditions on stage games and one-step transition probabilities.

2.3 Stochastic Potential Games and Stochastic Global Potential Games

We will now do a more in depth comparison between SPGs and SGPGs. We will see that, in a sense, SGPG's are *almost* a special case of SPGs. We begin by showing that there are SPGs that are not SGPGs.

Example 2.3.1. Consider a 2-player stochastic game with 3 states: s_1, s_2, and s_3 which initially starts in s_1. Both players have two actions a and b in each state. The stage games in s_1 and s_3 are trivial, that is, all actions yield zero utility for both players. The stage game in s_2 is given by the following bimatrix:

$$\begin{pmatrix} 4,2 & 4,2 \\ 2,1 & 2,3 \end{pmatrix}$$

Finally, the transition probabilities are

$$P_{12} = \begin{pmatrix} 0 & 0.5 \\ 0.5 & 1 \end{pmatrix}$$

and

$$P_{13} = \begin{pmatrix} 1 & 0.5 \\ 0.5 & 0 \end{pmatrix}$$

It is straightforward to see that these transition probabilities are modular. G_{s_1} and G_{s_3} are trivially potential games with potential function $\varphi \equiv 0$. G_{s_2} potential game with potential function:

$$\begin{aligned} \varphi(a,a) &= 0 \\ \varphi(a,b) &= 0 \\ \varphi(b,a) &= -2 \\ \varphi(b,b) &= 2 \end{aligned}$$

Since all stage games are potential games and the dynamics are modular, Γ is an SPG. Now, let

$$\begin{aligned} \pi^1 &= \pi^2 = [a,a,a] \\ \tilde{\pi}^1 &= [b,a,a] \\ \tilde{\pi}^2 &= [a,b,a] \end{aligned}$$

Then

$$\begin{aligned}
V^1_{s_1}(\tilde{\pi}^1, \pi^2) - V^1_{s_1}(\pi^1, \pi^2) &= 0.5 \\
V^2_{s_1}(\tilde{\pi}^1, \tilde{\pi}^2) - V^2_{s_1}(\tilde{\pi}^1, \pi^2) &= 0 \\
V^2_{s_1}(\pi^1, \tilde{\pi}^2) - V^2_{s_1}(\pi^1, \pi^2) &= 0 \\
V^1_{s_1}(\tilde{\pi}^1, \tilde{\pi}^2) - V^1_{s_1}(\pi^1, \tilde{\pi}^2) &= -1
\end{aligned}$$

Since these returns do not commute there cannot exist a global potential function. Notice that this example makes use of changing strategies in different states for different players. This displays the flexibility that SPGs have over SGPGs.

Now we give an example of a group of SGPGs that are not SPGs.

Example 2.3.2. Let Γ be a stochastic team game (ie all players receive the same utility in every state) with non-modular dynamics. By definition it cannot be an SPG. However, it admits a global potential function

$$\Phi(s, \pi) = V^i_\pi(s)$$

Note that the right hand side is independent of the choice of player i since all players receive the same utilities and hence have the same V functions.

Now that we've established that SPGs and SGPGs are distinct classes of stochastic games, we'll examine their relationship in more detail. For the remainder of the section we will only work with *layered* stochastic games. Furthermore, we introduce an additional property of stochastic games that will be useful:

Definition 2.3.3. An n-player stochastic game has the *termination property* if, for every state $s \in \mathcal{S}$ there exists a joint action $a^{term}_s \in \mathcal{A}^1 \times \cdots \times \mathcal{A}^n$ which provides zero utility for all players.

Theorem 2.3.4. *A layered n-player stochastic global potential game* Γ *with the termination property is either a stochastic potential game or a stochastic team game.*

Proof. Let Φ be the global potential function for Γ. Fix a state $s \in \mathcal{S}_k$ and joint action a in state s. Let π be the joint behavior with $\pi^i = a$ and $\pi'_s = a^{term}_{s'}$ for all other states. Let $b^i \in \mathcal{A}^i$ be an alternative action for player i in state s. The return for player i starting in state s is:

$$V^i_\pi(s) = u^i_s(\pi_s) + \sum_{s' \in \mathcal{S}_{k+1}} P_{ss'}(\pi_s) V^i_\pi(s')$$

Since $\pi_{s'} = a^{term}_{s'}$ for all $s' \neq s$, all of the $V^i_\pi(s')$ are zero. Hence

$$V^i_\pi(s) = u^i_s(\pi_s) = u^i_s(a)$$

Now, by assumption, changes in value at a state are aligned with changes in the potential function so that

$$\Phi(s, \pi \backslash b^i) - \Phi(s, \pi) = V^i_{\pi \backslash b^i}(s) - V^i_\pi(s) = u^i_s(a \backslash b^i) - u^i_s(a)$$

So the function

$$\varphi_s(a) = \Phi(s, \pi)$$

is a potential function for the stage game G_s. Hence Γ satisfies the first property of SPGs.

Now we must examine the transition probabilities of Γ. Consider a state s, a joint pure behavior π with $\pi_s = a$, players i and j, and $b^i \in \mathcal{A}^i$ and $b^j \in \mathcal{A}^j$ alternative actions for players i and j in state s. Since the stage game in state s is a potential game, we have that

$$\Phi(s, \pi \backslash b^i b^j) - \Phi(s, \pi \backslash b^i) + \Phi(s, \pi \backslash b^i) - \Phi(s, \pi) = \Phi(s, \pi \backslash b^i b^j) - \Phi(s, \pi \backslash b^i) + \Phi(s, \pi \backslash b^i) - \Phi(s, \pi)$$

implies that

$$\sum_{s' \in \mathcal{S}_{k+1}} \left[P_{ss'}(a \backslash b^i b^j) - P_{ss'}(a \backslash b$$

$$\sum_{s' \in \mathcal{S}_{k+1}} \left[P_{ss'}(a \backslash b^i b^j) - P_{ss'}(a \backslash b^j)\right] V^i_\pi(s') + \sum_{s' \in \mathcal{S}_{k+1}} \left[P_{ss'}(a \backslash b^j) - P_{ss'}(a)\right] V^j_\pi(s')$$

We can rearrange this to obtain a familiar relationship:

$$\begin{aligned}
&\sum_{s' \in \mathcal{S}_{k+1}} \left[P_{ss'}(a \backslash b^i b^j) - P_{ss'}(a \backslash b^i) - P_{ss'}(a \backslash b^j) + P_{ss'}(a)\right] V^j_\pi(s') \\
&= \sum_{s' \in \mathcal{S}_{k+1}} \left[P_{ss'}(a \backslash b^i b^j) - P_{ss'}(a \backslash b^i) - P_{ss'}(a \backslash b^j) + P_{ss'}(a)\right] V^j_\pi(s')
\end{aligned}$$

Notice that this equation resembles [equation for continuation game] except the continuation vectors z^i are replaced by expected rewards $V^i_\pi(s)$. With arbitrary z^i, we could construct them such that the only way to satisfy the equations was to have modular dynamics. In contrast, the $V^i_\pi(s)$ are given as part of the stochastic game, and span a comparatively low-dimensional subspace. This is most readily seen in the case of team games, where $V^i_\pi(s) = V^j\pi(s)$ for all i and j, so that they span a one-dimensional space.

We can add a definition / assumption about this. It would amount to something a bit stronger than the termination property, which is something like: 'There are joint behaviors that give one player reward one and all others reward zero.'

□

2.3.1 Further investigation of global potentials

Notice that in the preceding analysis, we made use of consistency equations [reference eq] for SGPGs in which the changes in player i and player j's behaviors occurr in the same state s. However, if we vary the states that the behavior changes occur in, we can derive a much larger set of consistency equations that restrict the form of SGPGs even further.

Suppose alternative actions b^i and b^j occur in adjacent layers, that is, in states $s_1 \in \mathcal{S}_k$

and $s_2 \in \mathcal{S}_{k+1}$ respectively. Fix a baseline joint pure behavior π, and for notational simplicity let $a_1 = \pi_{s_1}$ and $a_2 = \pi_{s_2}$. As usual, we start from the consistency equation

$$\Phi(s_1, \pi\backslash b^i b^j) - \Phi(s_1, \pi\backslash b^i) + \Phi(s_1, \pi\backslash b^i) - \Phi(s_1, \pi) = \Phi(s_1, \pi\backslash b^i b^j) - \Phi(s_1, \pi\backslash b^i) + \Phi(s_1, \pi\backslash b^i) - \Phi(s_1, \pi)$$

Replacing each difference with a difference in values, and then simplifying yields:

$$\begin{aligned} \Phi(s_1, \pi\backslash b^i b^j) - \Phi(s_1, \pi\backslash b^i) = \quad & u^j_{s_1}(a_1\backslash b^i) + \sum_{s' \in \mathcal{S}_{k+1}} P_{s_1 s'}(a_1\backslash b^i) V^j_{\pi\backslash b^i b^j}(s') \\ - u^j_{s_1}(a_1\backslash b^i) - \sum_{s' \in \mathcal{S}_{k+1}} P_{s_1 s'}(a_1\backslash b^i) V^j_{\pi\backslash b^i b^j}(s') & \end{aligned}$$

Notice that changing actions in earlier layer states has no effect on the value of later states so

$$V^j_{\pi\backslash b^i b^j}(s') = V^j_{\pi\backslash b^j}(s')$$

Furthermore, when $s' \neq s_2$,

$$V^j_{\pi\backslash b^j}(s') = V^j_{\pi}(s')$$

So most terms above cancel, leaving

$$\Phi(s_1, \pi\backslash b^i b^j) - \Phi(s_1, \pi\backslash b^i) = P_{s_1 s_2}(a_1\backslash b^i) \left[Q^j_{\pi}(s_2, a_2\backslash b^j) - Q^j_{\pi}(s_2, a_2) \right]$$

Following a similar process for the other differences in the consistency equation we get

$$\Phi(s_1,\pi\backslash b^i) - \Phi(s_1,\pi) = u^i_{s_1}(a_1\backslash b^i) - u^i_{s_1}(a) + \left[P_{s_1s_2}(a_1\backslash b^i) - P_{s_1s_2}(a)\right] V^i_\pi(s_2)$$

$$\Phi(s_1,\pi\backslash b^ib^j) - \Phi(s_1,\pi\backslash b^j) = u^i_{s_1}(a_1\backslash b^i) - u^i_{s_1}(a) + \left[P_{s_1s_2}(a_1\backslash b^i) - P_{s_1s_2}(a)\right] Q^i_\pi(s_2,a_2\backslash)$$

$$\Phi(s_1,\pi\backslash b^j) - \Phi(s_1,\pi) = P_{s_1s_2}(a_1)\left[Q^j_\pi(s_2,a_2\backslash b^j) - V^j_\pi(s_2)\right]$$

Substituting these into the original consistency equation and rearranging terms results in

$$\begin{aligned}&\left(P_{s_1s_2}(a) - P_{s_1s_2}(a\backslash b^i)\right)\left(Q^i_\pi(s_2,a_2\backslash b^j) - Q^i_\pi(s_2,a_2)\right)\\ = &\left(P_{s_1s_2}(a) - P_{s_1s_2}(a\backslash b^i)\right)\left(Q^j_\pi(s_2,a_2\backslash b^j) - Q^j_\pi(s_2,a_2)\right)\end{aligned}$$

** Add an analysis about what this means. Close to a team game **

2.4 Equilibria

Before we study learning dynamics it is important that we first confirm the existence and characterize the properties of Nash equilibria in both SPGs and SGPGs. In the normal-form game setting existence can be taken for granted since normal-form games always admit Nash equilibria. However, a general existence theorem does not exist for stochastic games. In fact it has been demonstrated that there are stochastic games that admit *no Nash equilibria* [cite]. We will show that SPGs and SGPGs both admit pure Nash equilibria.

Theorem 2.4.1. *If* Γ *is a stochastic global potential game then* Γ *admits at least one pure Nash equilibria. In particular, any joint behavior that maximizes the global potential function is a Nash equilibrium.*

Proof. Let π be a pure joint behavior that maximizes Φ, the global potential of Γ. If τ^i is an alternative behavior for player i, we know that, from any state s, we have

$$\Phi(s,\pi) \geq \Phi(s,\pi\backslash\tau^i$$

and therefore

$$V^i_\pi(s) \geq V^i_{\pi\backslash\tau^i}(s)$$

So π is a Nash equilibrium. $\square$

Theorem 2.4.2. *if* Γ *is a layered stochastic potential game then* Γ *admits at least one pure Nash equilibrium.*

Proof. Let Γ be an n-player layered stochastic potential game. As usual denote the state partition by $\mathcal{S}_1,\ldots,\mathcal{S}_T$, the stage potential games by G_s with potential φ_s and utilities u^i_s.

We will construct a pure Nash equilibrium π by a backwards iterative method through the partitions.

First, for each state $s \in \mathcal{S}_T$ let a_s be a pure Nash equilibrium for the stage game G_s. Set $\pi_s := a_s$ and then $V^i_\pi(s) = u^i_s(a_s)$. Note that we haven't fully defined π yet, but it is okay to talk about $V^i_\pi(s)$ for states $s \in S_T$ since we have described the behavior of π at layer T.

Next, for each state $s \in \mathcal{S}_{T-1}$ consider the continuation game $G_s(V^i_\pi)$ with utilities

$$u^i_s(a) + \sum_{s'\in\mathcal{S}_T} P_{ss'}(a)V^i_\pi(s')$$

Since G is a stochastic potential game, each $G_s(V^i_\pi)$ is a potential game and hence admits a pure Nash equilibrium a_s. For the states in $\mathcal{S}_{T-1}$ set $\pi_s := a_s$ and then $V^i_\pi(s) = u^i_s(a_s) + \sum_{s'\in\mathcal{S}_T} P_{ss'}(a_s)V^i_{s'}$.

Iterating this process, for each state $s \in S_j$ consider the continuation game $G_s(V^i_\pi)$ with utilities

$$u^i_s(a) + \sum_{s'\in\mathcal{S}_T} P_{ss'}(a)V^i_\pi(s')$$

This game admits a pure Nash equilibrium a_s. We set $\pi_s := a_s$ for all $s \in \mathcal{S}_j$ and $V^i_\pi = u^i_s(a_s) + \sum_{s'\in\mathcal{S}_T} P_{ss'}(a_s)V^i_\pi(s')$.

Eventually this process terminates, at which point it defines a pure joint strategy in every state ie a pure joint behavior π. Next we need to show that π is a Nash equilibrium.

Fix a player i whose behavior under π is π^i and consider an alternative behavior τ^i. Let t_0 be the last layer where π^i and τ^i differ. That is, the largest t_0 such that there exists $s_0 \in \mathcal{S}_{t_0}$ with$\pi^i_{s_0} \neq \tau^i_{s_0}$. Then the behaviors π^i and $\pi\backslash\tau^i$ correspond to strategies in the game $G_{s_0}(z(t_0+1))$. But, since we know that π_s is a Nash equilibrium for the game $G_{s_0}(V^i_\pi)$, it must be the case that

$$u^i_s(\pi_s) + \sum_{s' \in \mathcal{S}_{t_0+1}} P_{ss'}(\pi_s) V^i_\pi(s') \geq u^i_s(\pi_s\backslash\tau^i) + \sum_{s' \in \mathcal{S}_{t_0+1}} P_{ss'}(\pi_s\backslash\tau^i) V^i_\pi(s')$$

Therefore π is a Nash equilibrium.

□

2.5 Learning

In this final section we will focus on defining learning algorithms that converge to Nash equilibria in SGPGs and SPGs. In particular we will define a stochastic game version of log linear learning. We will show that this converges in SGPGs but not necessarily in SPGs. Finally we will introduce two variants that converge in SPGs.

2.5.1 Log-linear Learning in Repeated Normal-form Games

Log-linear learning is a learning algorithm that provably converges to pure Nash equilibria in potential games. That is, if players repeatedly play the same normal-form game, adjusting their strategies at each timestep according to the log-linear learning algorithm, their play will approach a pure Nash equilibrium in probability. In fact, one can say even more: log-linear learning will converge to *potential maximizing* pure Nash equilibria. Authors ofter refer to this as an "equilibrium selection property" in that the algorithm discriminates between Nash equilibria with differing properties. In the context of potential games, the potential function often characterizes some notion of social/global utility, and so it is particularly useful to have an algorithm that will converge to such behavior.

Log-linear learning proceeds as follows. At each timestep t, players simultaneously select actions $a^i \in \mathcal{A}^i$ and receive utility $u^i(a)$. The players select these actions according to probability distributions $x^i \in \mathcal{A}^i$ over their action sets. The players update these distributions as follows. At time t, one player i is selected uniformly at random and generates a distribution x^i_t. All other players will play the same action they did at the previous timestep, ie $a^j_t = a^j_{t-1}$ for players $j \neq i$. In contrast, player i will play by sampling from the distribution

$$x^i_t(a) = \frac{e^{u^i(a, a^{-i}_{t-1})}/\tau}{\sum_{b \in \mathcal{A}^i} e^{u^i(b, a^{-i}_{t-1})}/\tau}$$

where a^{-i}_{t-1} is the list of actions played at time $t-1$ by players other than player i, $x^i_t(a)$ is the probability of selecting action a under the distribution x^i_t, and τ is a *temperature parameter*. Notice that as the temperature parameter tends to zero, the probability distribution concentrates on best responses to the other players' actions at the last timestep. In [cite Blume statistical mechanics of strategic interaction] it is shown that the stationary distribution μ of joint strategies is

$$\mu(a) = \frac{e^{\varphi(a)/\tau}}{\sum_b e^{\varphi(b)/\tau}}$$

where b varies over the set of joint actions $b \in \times_i \mathcal{A}^i$.

From this explicit form of the stationary distribution it follows that as the temperature is brought to zero, the joint distribution concentrates on potential maximizing joint behaviors.

In [cite Marden revisiting log-linear learning, or maybe the paper they cite] the authors provide an alternative proof method using the theory of resistance trees in Markov processes. This allows the authors to characterize the stochastically stable states (which are potential maximizing joint actions), without explicitly writing down the stationary distribution.

* * Copied from "Revisiting log linear learning"

Let P^0 denote the probability transition matrix for a finite state Markov chain over the state space Z. We refer to P^0 as the unperturbed process. Consider a perturbed process where the size of the perturbation is indexed by a scalar $\epsilon > 0$, and let P^ϵ be the associated transition matrix. The process P^ϵ is called a regular perturbed Markov process if P^ϵ is ergodic for sufficiently small ϵ and P^ϵ approaches P^0 at an exponentially smooth rate. Specifically, the latter condition means that for all $z, z' \in Z$,

$$\lim_{\epsilon \to 0^+} P^\epsilon_{z \to z'} = P^0_{z \to z'}$$

and,

$$P^\epsilon_{z \to z'} > 0 \text{ for some } \epsilon > 0 \text{ implies } \lim_{\epsilon \to 0^+} \frac{P^\epsilon_{z \to z'}}{\epsilon^{R(z \to z'}}$$

for some nonnegative real number $R(z \to z')$, which we call the *resistance* of the transition $z \to z'$ under the perturbed process.

Construct a complete directed graph with $|Z|$ vertices for each state. The vertex corresponding to state z_j will be called j. The weight on the directed edge $i \to j$ is the resistance $R(z_i \to z_j)$. A j-tree, or "tree rooted at j", T, is a directed tree such that all paths terminate at j. The resistance of T is the sum of the edge resistances composing it, and the *stochastic potential*, γ_j, of z_j is the minimum resistance amongst all j-trees.

Theorem 2.5.1 (cite Evolution of convention)**.** *Let P^ϵ be a regular perturbed Markov process, and for each $\epsilon > 0$ let μ^ϵ be the unique stationary distribution of P^ϵ. Then* $\lim_{\epsilon \to 0^+} \mu_\epsilon$ *exists and the limiting distribution μ^0 is a stationary distribution of P^0. The stochastically stable states (i.e. the support of μ^0) are precisely those states with minimal stochastic potential. Furthermore, if a state is stochastically stable then the state must be in a recurrent class of the unperturbed process P^0.*

* *

In [Marden log-linear learning] the authors show that log-linear learning converges to potential maximizing behavior by proving the following sequence of results:

Lemma 2.5.2 (Marden 3.1)**.** *Log-linear learning induces a regular perturbed Markov process where the unperturbed Markov process is an asynchronous best reply process and the resistance of any feasible transition $a \to b = (b^i, a^{-i})$ is*

$$R(a \to b) = \max_{a^i_* \in \mathcal{A}^i} u^i(a^i_*, a^{-i}) - u^i(b)$$

Lemma 2.5.3 (Marden 3.2)**.** *Consider any finite n-player potential game with potential function* $\varphi : \mathcal{A} \to \mathbb{R}$ *where all players adhere to log-linear learning. For any feasible action path*

$$\mathcal{P} = \{a^0 \to a^1 \to \cdots \to a^m\}$$

and its reverse path

$$\mathcal{P}^R = \{a^m \to a^{m-1} \to \cdots \to a^0\}$$

the difference in the total resistance across the paths is

$$R(\mathcal{P}) - R(\mathcal{P}^R) = \varphi(a^0) - \varphi(a^m)$$

and finally,

Proposition 2.5.4 (3.1 Marden)**.** *Consider any finite n-player potential game with potential function* $\varphi : \mathcal{A} \to \mathbb{R}$ *where all players adhere to log-linear learning. The stochastically stable states are the set of potential maximizers, i.e.,* $\{a \in \mathcal{A} : \varphi(a) = \max_{a^* \in \mathcal{A}} \varphi(a^*)\}$

2.5.2 A Generalization to Stochastic Games

This section will generalize log-linear learning to the stochastic setting and show that, in the case of stochastic global potential games, the stochastically stable states of generalized log-linear learning are precisely those joint behaviors that maximize the global potential function.

Generalized log-linear learning proceeds as follows. At each timestep t, players select pure behaviors π^i. The players update their behavior choices as follows. At time t, a player i and a state s are selected uniformly at random. Player i will generate a distribution x^i_s. All other players select the same behavior as they did at the previous timestep, i.e. $\pi^j_t = \pi^j_{t-1}$ for players $j \neq i$. In contrast, player i will play by sampling from the distribution

$$x^i_s(a) = \frac{e^{Q^i_\pi(s,a,a^{-i}_{t-1})}/\tau}{\sum_{b \in \mathcal{A}^i} e^{Q^i_\pi(s,b,a^{-i}_{t-1})}/\tau}$$

where a_{t-1}^{-i} is the action played in state s at time $t-1$ by all players except player i. $x_s^i(a)$ is the probability of selecting action a in state s under the distribution x_s^i, and τ is a *temperature parameter*.

It is important to note that this learning method is not payoff-based. It requires that players are able to compute their Q-values in order to update their strategy.

Lemma 2.5.5. *Generalized log-linear is a regular perturbed Markov process where the resistance of any feasible transition $\pi \longrightarrow \tau = (a_s^i, \pi^{-i})$ is*

$$R(\pi \longrightarrow \tau) = \max_{a_*^i \in \mathcal{A}^i} Q_\pi^i(a_*^i, \pi^{-i}) - Q_\pi^i(s, \tau)$$

Proof. The unperturbed process behaves as follows for each timestep $t > 0$. Choose a player i and state s, both uniformly at random. Change player i's action in that timestep to an element of $a_* \in \arg\max Q_{\pi_{t-1}}^i(s, a, \pi_s^{-i})$. Keep the behavior of player i in all other states unchanged, and keep the behaviors of all other players the same.

Under the perturbed process P^ϵ, the probability of transitioning from π to τ is

$$P_{\pi \to \tau}^\epsilon = \frac{1}{n} \frac{1}{|\mathcal{S}|} \frac{\epsilon^{-Q_\pi^i(s, a_s^i, \pi_s^{-i})}}{\sum_{b \in \mathcal{A}^i} \epsilon^{-Q_\pi^i(s, b, \pi_s^{-i})}}$$

Let $W_i(s, \pi) = \max_{a_*^i \in \mathcal{A}^i} Q_\pi^i(a_*^i, \pi^{-i})$. Multiply the numerator and denominator by $\epsilon^{W_i(s,\pi)}$ yields

$$P_{\pi \to \tau}^\epsilon = \frac{1}{n} \frac{1}{|\mathcal{S}|} \frac{\epsilon^{W_i(s,\pi) - Q_\pi^i(s, a_s^i, \pi_s^{-i})}}{\sum_{b \in \mathcal{A}^i} \epsilon^{W_i(s,\pi) - Q_\pi^i(s, b, \pi_s^{-i})}}$$

So then,

$$\lim_{\epsilon \to 0^+} \frac{P_{\pi \to \tau}^\epsilon}{\epsilon^{W_i(s,\pi) - Q_\pi^i(s, a_s^i, \pi_s^{-i})}} = \lim_{\epsilon \to 0^+} \frac{1}{n|\mathcal{S}|} \frac{1}{\sum_{b \in \mathcal{A}^i} \epsilon^{W_i(s,\pi) - Q_\pi^i(s, b, \pi_s^{-i})}}$$

The summands in the denominator will approach either 0 or 1 depending on if b is in $\arg\max Q_{\pi_{t-1}}^i(s, a, \pi_s^{-i})$. So this limit is equal to

$$\frac{1}{n|\mathcal{S}||\arg\max Q_{\pi_{t-1}}^i(s, a, \pi_s^{-i})|}$$

Therefore generalized log-linear learning is a regular perturbed Markov process with resistances

$$R(\pi \to \tau) = \max_{a^i_* \in \mathcal{A}^i} Q^i_\pi(a^i_*, \pi^{-i}) - Q^i_\pi(s, \tau)$$

□

Should define feasible paths here

Lemma 2.5.6. *Consider any finite n-player SGPG with potential function* $\Phi : \mathcal{A} \to \mathbb{R}$ *where all players adhere to log-linear learning. For any feasible action path*

$$\mathcal{P} = \{\pi_0 \to \pi_1 \to \cdots \to \pi_m\}$$

and its reverse path

$$\mathcal{P}^R = \{\pi_m \to \pi_{m-1} \to \cdots \to \pi_0\}$$

the difference in the total resistance across the paths is

$$R(\mathcal{P}) - R(\mathcal{P}^R) = \varphi(\pi_0) - \varphi(\pi_m)$$

Proof. Consider a single edge $\pi_k \to \pi_{k+1}$ in the path and its reverse $\pi_{k+1} \to \pi_k$. By the previous lemma we have

$$R(\pi_k \to \pi_{k+1}) = W_i(s, \pi_k) - Q^i_{\pi_k}(s, a^i_{k+1}, (\pi^{-i}_k)_s) R(\pi_{k+1} \to \pi_k) = W_i(s, \pi_{k+1}) - Q^i_{\pi_{k+1}}(s, a^i_k, (\pi^{-i}_{k+1})_s)$$

Since π_k and π_{k+1} only differ by player i's action in state s we have that:

$$W_i(s, \pi_{k+1}) = W_i(s, \pi_k) Q^i_{\pi_{k+1}}(\cdot) = Q^i_{\pi_k}(\cdot)(\pi^{-i}_{k+1})_s = (\pi^{-i}_k)_s$$

So then

$$R(\mathcal{P}) - R(\mathcal{P}^R) = Q^i_{\pi_k}(s, a^i_k, (\pi^{-i}_k)_s) - Q^i_{\pi_k}(s, a^i_{k+1}, (\pi^{-i}_k)_s) = \Phi(s, \pi_k) - \Phi(s, \pi_{k+1})$$

□

Proposition 2.5.7. *Consider any finite n-player GSPG with potential function* $\Phi : \mathcal{A} \to \mathbb{R}$ *where all players adhere to log-linear learning. The stochastically stable states are the set of potential maximizers, i.e.,* $\{\pi : \Phi(\pi) = \max_{\pi_*} \Phi(\pi_*)\}$

Proof. □

2.5.3 Convergent Variants for Stochastic Potential Games

Feasible paths only travel backwards through layers. As long as behavior eventually settles to a particular equilibrium the learning algorithm should work.

Chapter 3

Regret Matching in Stochastic Games