Instructing a Reinforcement Learner

Abstract

In reinforcement learning (RL), rewards have been considered the most important channel for understanding an environment's dynamics and have been very effectively used as a feedback mechanism. However, recently there have been interesting forays into other modes of understanding the environment. Using sporadic supervisory inputs is one such alternative. This brings into the learning process rich information about the world of interest. In this paper, we model these supervisory inputs as instructions, provide a mathematical formulation for the same and propose a framework to incorporate them into the learning process.

Introduction

Reinforcement Learning (RL) is a machine learning paradigm that closely resembles the human way of learning. It is a control based learning approach that tunes its control system based on evaluative feedback received from the environment. The learner interacts with the environment by taking actions. It receives feedback from the environment in the form of rewards and observations(states). The goal of the controller is to maximize the rewards collected in the long run. Unlike supervised learning, this feedback is partial. If the feedback was complete, the controller would be told the best action in a given state. Instead, a reinforcement learner explores the various actions it can perform, estimates the expected reward of choosing the action and accordingly learns the set of optimal state-action pairs.

Even though the trial-and-error approach of RL seems similar to the human learning process, there are important differences between the two. Humans benefit from prior knowledge of the environment, past experiences and inputs from other humans. An RL agent is devoid of access to any of these, resulting in inordinate amounts of exploration and long learning periods. There have been various approaches to speed up learning such as using deixis (Chapman 1991), hierarchical learning (Barto and Mahadevan 2003) and learning from humans (Rosenstein and Barto 2004; Atkeson and Schaal 1997; Maclin and Shavlik 1998; Clouse and Utgoff 1992).

In this paper, we concentrate on a specific technique of learning from humans called learning through instructions(Chapman 1991). Earlier methods either imitate the human in some manner or observe the human for prolonged durations in order to learn more about the regular domain dynamics . We propose a novel framework which allows the learner to understand the domain beyond rewards and transitions and facilitates exploitation of the implicit information such as structure in the state space and underlying patterns governing the domain. Additionally, our approach can learn well even with minimal interaction with humans.

Consider a human learning to throw a ball into a basket. The evaluative feedback will depend on how far the ball misses the target by. Whereas, instructive feedback will be a coach instructing him to *throw harder or slower*. Instructions could be of various forms. For example, consider the agent searching for a key. The agent could be instructed to "look in the key stand". The effect of this instruction is to reduce the agent's search space considerably. Take the case of an agent navigating an obstacle course. When it is obstructed by a puddle of water, the agent is instructed to "jump". This instruction can then be reused by generalizing it over puddles of various locations,liquids, colors, shapes, etc. Thus efficiently using the information in the instruction.

In this paper, we incorporate learning from such instructions into traditional RL methods. We give a mathematical formulation for instructions and outline two kinds of instructions, π-instructions and Φ-instructions. We also provide algorithms that utilize both instructive and evaluative feedback. These are then empirically shown to perform better than traditional RL methods.

In the next section, we discuss some existing literature that have similar motivations. We then present the necessary notations and background required to define instructions. We then introduce instructions mathematically and discuss two types of instructions. In subsequent sections, we give a framework for exploiting these two types of instructions along with the experimental evaluation. The experiments are performed on two domains, one for each type of instruction. Finally we analyze our approach paying attention to interesting facets of our approach and presenting directions of future research.

Related Work

In this section, we discuss those approaches that we feel are closely related to our work.

Inverse Reinforcement Learning Informally, the Inverse Reinforcement Learning (IRL) Problem (Ng and Russell 2000) can be considered as the task of determining the reward function being optimized given measurements of the agent's behavior over time in various situations, measurements of its sensory inputs if necessary, and a model of the world, if available. Motivation for this approach lies in the presupposition that the reward function is the most precise and transferable definition of the task and that the reward functions of most real world problems cannot be modeled completely resulting in the need to observe experts.

Implicit Imitation Accelerating Reinforcement Learning through Implicit Imitation (Price and Boutilier 2003) proposes the Implicit Imitation model. This model allows the agent to observe an expert's behavior and use the observed state transitions to update its estimates of state values and actions. The agent does not explicitly imitate the trajectory of the expert. In addition to learning the domain dynamics, the agent can also take cues from the expert in terms of regions of state space worth exploring etc.

Unlike in IRL and Implicit Imitation, the approach we propose does not require the agent to observe an expert's behavior over several episodes. Instead the expert gives occasional inputs in the form of instructions only. Neither do we look to optimize an agent's behavior based on an estimated reward function as in IRL, nor do we incorporate observed dynamics of another agent as in Implicit Imitation. Instead, we generalize the information contained in an instruction over the entire search space. We carefully make use of this generalization to speed up learning. In general, we look beyond the traditional reward function and transition dynamics as will be explained in later sections.

In spirit, our approach is similar to the above two since all three approaches attempt to model an expert in their own way and use this model whenever necessary.

Notation

Problems in RL are generally modeled as *Markov Decision Processes* (MDPs). The MDP framework forms the basis of our definition of instructions.

A MDP is a tuple $\langle S, A, \psi, P, R\rangle$, where S is a finite set of states, A is a finite set of actions, $\psi \subseteq SXA$ is the set of admissible state-action pairs, $P : \psi \rightarrow [0, 1]$ is the transition probability function with $P(s, a, s')$ being the probability of transition from state s to state s' by performing action a. $R : \psi \rightarrow \mathbb{R}$ is the expected reward function with $R(s, a)$ being the expected reward for performing action a in state s (this sum is known as return). $A_s = \{a|(s, a) \in \psi\} \subseteq A$ be the set of actions admissible in state s. We assume that A_s is non-empty for all $s \in S$. $\pi : \psi \rightarrow [0, 1]$ is a stochastic policy, such that $\forall s \in S \sum_{a\in A_s} \pi(s, a) = 1$. $\forall(s, a) \in \psi$, $\pi(s, a)$ gives the probability of executing action a in state s.

The value of a state-action pair conditioned on policy π, $Q_\pi(s, a)$, is the expected value of a sum of discounted future rewards of taking action a, stating from state s, and following policy π thereafter. The optimal value functions assign to each state-action pair, the highest expected return achievable by any policy. A policy whose value function is optimal is an optimal policy π^*. Conversely, for any stationary MDP, any policy greedy with respect to the optimal value functions must be an optimal policy : $\pi^*(s) = argmax_a Q^*(s, a) \forall s \in S$, where $Q^*(s, a)$ is the optimal value function. If the RL agent knew the MDP, it could be able to compute the optimal value function, and from it extract the optimal policy. However, in the regular setting, the agent is only aware of ψ, the state-action space and must learn Q^* by exploring. The Q-learning algorithm learns the optimal value function by updating its current estimate,$Q_k(s, a)$, of $Q^*(s, a)$ using this simple update (Barto and Mahadevan 2003),

$$Q_{k+1}(s, a) = Q_k(s, a)+\alpha\big[r+\gamma max_{a'} Q_k(s', a')-Q_k(s, a)\big] \quad (1)$$

α is the learning rate of the algorithm, $\gamma \in [0, 1]$ is the discounting factor and a' is the greedy action in s'.

An option (or a temporally extended action)(Barto and Mahadevan 2003) is described by the tuple $\mathcal{O} = \langle I, \pi, \beta\rangle$, where the $I \subseteq S$ is an initiation set, π is the option policy, and $\beta : S \rightarrow [0, 1]$ is the termination function. An option can be invoked in any state $s \in I$, following which, the agent will execute the policy π until it terminates as modeled by $\beta(s)$.

Q-learning applies to options too and is referred to as SMDP Q-learning (Barto and Mahadevan 2003).

$$Q_{k+1}(s, o) = (1-\alpha)Q_k(s, o)+\alpha\big[R+\gamma^\tau max_{o'} Q_k(s', o')\big] \quad (2)$$

where R is the sum of the time discounted rewards accumulated while executing the option and τ is the time taken for execution.

Structured States

Understanding the effect of instructions is easy if the states are structured. In the next section, we motivate the use of such instructions using simple operations on the structured state space explained here.

The set of states S is *structured* by representing it as a cross-product of an indexed family $\{S_\alpha|\alpha \in D\}$, where D is the set of state features (Zeigler 1972). In general $\alpha \in D$ is referred to as the coordinate and S_α is its state set. The structure assignment is a one-one mapping from S to $\prod_{\alpha\in D} S_\alpha$. Thus, a state $s \in S$ is represented as $(s_{\alpha_1}, s_{\alpha_2}, \ldots, s_{\alpha_i}, \ldots)$ where s_{α_i} is a value of the feature set S_{α_i}.

Let f be a set of indexed functions such that $\{f_i : S \rightarrow B_i|i \in E\}$, where E is a different set of coordinates and B_i is the corresponding state set. The cross product function $\prod_{i\in E} f_i : S \rightarrow \prod_{i\in E} B_i$ is defined by $\prod_{i\in E} f_i(s) = (f_1(s), f_2(s), \ldots)$. In this paper we use this cross product function which is a special case of a structured function.

A special class of indexical functions operating on the structured set S are coordinate projections $\{\rho_\alpha|\alpha \in D\}$ where $\rho_{\alpha_i} : S \rightarrow S_{\alpha_i}$ such that $\rho_{\alpha_i}(s_{\alpha_1}, s_{\alpha_2}, \ldots, s_{\alpha_i}, \ldots) = s_{\alpha_i}$. Extending the above cross product function to projections of S : For $D' \subseteq D, \rho_{D'} : S \rightarrow \prod_{\alpha\in D'} S_\alpha$ is given by $\rho_{D'} = \prod_{\alpha\in D'} \rho_\alpha$. For example, $\rho_{\{\alpha_2,\alpha_4\}}(s_{\alpha_1}, s_{\alpha_2}, \ldots) = (s_{\alpha_2}, s_{\alpha_4})$.

Every subset $D' \subseteq D$ induces a partition $K_{D'}$ on S such that two states $s, s' \in S$ belong to the same block B_i only if: $\rho_{D'}(s) = \rho_{D'}(s')$ and is denoted by

$$[s]_{K_{D'}} = [s']_{K_{D'}} \quad (3)$$

Instructions

In general, any external input to the RL agent, that it can use to make decisions or direct its exploration or modify its belief in a policy can be called advice. If the advice is inviolate i.e., the learner cannot override it, it is called an *Instruction*. For example, an agent that is learning to cut vegetable can be instructed to *use the sharp edge of the knife*.

Instructions have also been used to specify regions of the state space or objects of interest. For example, in his thesis (Chapman 1991), Chapman gives Sonja, a game player, the instruction, *Get the top most amulet from that room*. This instruction binds the region of search and the object of interest without divulging any information about how to perform the task.

Mathematical Formulation

The policy $\pi(s, a)$ can be represented as :

$$\pi(s, a) = G(\Phi(s), a) \quad (4)$$

where $\Phi(s)$ models operations on the state space. $G(.)$ is a mapping from $(\Phi(s), a)$ to the real interval $[0, 1]$. $\Phi(s)$ can either model mappings to a subspace of the current state space or model projections of the state s onto a subset of features. This representation of the policy makes it easy to understand the two types of instructions explained below.

π-Instructions Instructions of this type are in the form of action or option to be performed at the current state : $I_\pi(s) = a$, where $a \in A_s$. The effect of these instructions is to directly modify the policy $\pi(s, a) = 1$. As an example, consider an RL agent learning to maneuver through a farm and is in a state with a puddle in front (s_{puddle}). The instruction *jump* is incorporated as $\pi(s_{puddle}, jump) = 1$. We use such instructions in the experiments on the Transporter domain.

Φ-Instructions Instructions of this type are given as structured functions denoted by I_Φ. In this paper, we only use structured functions that are projections. Such an instruction, I_Φ would be captured by $\Phi(s)$ as $\rho_{D'}(s), D' \subseteq D$. D is the set of features representing the state set S and $\rho_{D'}$ is the projection operation mentioned in section . $D' \subseteq D$ captures the possibility that some features in a state representation are redundant or uninfluential. For example, consider an RL agent learning to throw balls. The instruction, *"Ignore the ball's color"* will be incorporated as $D' = D - \{ballcolor\}$. We use similar instructions in the experiments on the Deictic Game domain.

Transporter Domain

In this domain we provide the learner with π-instructions as described in the previous section and give a framework for exploiting these instructions. The layout of the domain

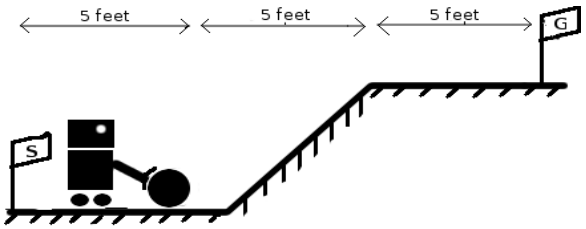


Figure 1: Transporter Domain

is shown in Fig 1. The task of the RL agent is to transport an object from the starting position to the goal position. The object can be a sphere, a cube or a cylinder weighing between 0 and 15 pounds. The path to the goal is 15 feet long. The first and last 5 feet are on level floor and the remaining are on a slope. The agent can transport an object by invoking 1 of 3 options. The options are *carry using 1 arm*, *carry using 2 arms* and *push*. All options move the agent towards the goal for a maximum of 5 feet.

The dynamics of the options depend on the shape and weight of the object as well as the slope of the floor. Heavier objects take longer time to be transported. *Carry using 1 arm* is faster than *carry using 2 arms*, which in turn is faster than *push along the floor*. All options take more time on the ramp. *Pushing* a sphere or cylinder is faster than *pushing* a cube. Also, the time taken to pick up an object on the floor is proportional to its weight. An option may not execute to completion always. The agent might drop the object halfway, depending on the object properties and the slope of the floor. For example, a heavy object is dropped more easily than a light object. Similarly, carrying a cube using 2 arms is safer than carrying a cube with 1 arm. Also, the probability of dropping an object is greater on the slope. The exact dynamics of the domain along with the implementation will be made available at the online RLGlue repository at a later date.

A learner would optimally *carry light objects using 1 arm*, *push heavy objects*, *carry cubes using 2 arms*, *push a sphere or a cylinder* and *carry objects using 2 arms on the slope*. The state features observed by the agent are $\langle obj - shape, obj - weight, current - position, obj - in - arm \rangle$, where $obj - in - arm$ indicates whether the object is being carried or is on the floor.

Learning Algorithm

The standard RL approach uses SMDP-Q learning with an ϵ greedy exploration policy. The reward function used is the minimally informative and is described simply as $reward = -1$ till termination. Our algorithm is present in Algorithm 1.

Occasionally(with some probability ζ), we provide the agent with π - instructions i.e., tell the agent the best option to perform in a given state. The agent generalizes over these instructions by using a standard classifier. This classifier outputs an option based on the given state. The set of all π - instructions seen so far forms the training data for this classifier. In this particular implementation we have used a k-NN classifier. Every time an instruction is given, $\{s, I_\pi(s)\}$ is added to the dataset $\mathcal{D}_\mathcal{I}$. The flow of instructions is cut-off

after a fixed number of episodes. Regular Q function updates continue to take place on the side as in a standard Q-learner.

At every decision point , the agent chooses between the option recommended by the k-NN model and the Q-learner by comparing their confidences. The confidence of the k-NN model is computed as the inverse squared distance between the given state and its nearest neighbor (of the same class as predicted by the k-NN). The variance in the Q function is used to represent the confidence measure of the Q-learner. If the variance in the value of a particular state-option pair is very low, it implies that the value has converged to the final value defined by the policy. In other words, the confidence of the Q-learner in an option is inversely proportional to the variance of the Q function at that state-option pair.

Whenever the confidence of the k-NN model is high (as decided by a threshold) and the confidence of the Q-learner is low, it performs the action suggested by the k-NN model. Otherwise, it follows the policy represented by $Q(s,a)$. We assume that eventually the Q-learner might become more optimal than the k-NN model. This might be due to errors in generalization or a faulty instructor. Hence the Q-learner is given the benefit of the doubt. The threshold parameters $Qthresh$ and $Cthresh$ have to then be tuned.

Algorithm 1 LearnWithπInstructions($\mathcal{D}_{\mathcal{I}}$)

while *episode not terminated* **do**
  s is the current state
  if $I_\pi(s)$ *available* **then**
    $a \leftarrow I_\pi(s)$
    $\mathcal{D}_{\mathcal{I}} \leftarrow \mathcal{D}_{\mathcal{I}} \cup \{s, I_\pi(s)\}$
  else
    if $\texttt{conf}(Q(s, \arg\max_b Q(s,b)) < Qthresh$ **and** $\texttt{conf}(kNN(s)) > Cthresh$ **then**
      $a \leftarrow$ *kNN-Classify*$(s; \mathcal{D}_{\mathcal{I}})$
    else
      $a \leftarrow \arg\max_b Q(s,b)$
    end if
    Perform option a
    Update $Q(s,a)$
  end if
end while

Analysis

Learning the optimal policy on this domain is hard due to the complex dynamics and variety in the objects. The performance of our approach is shown in Fig 2[1]. Our approach converges at around 2000 episodes, whereas standard SMDP Q learning takes more than 10000 episodes to show similar performance. In order to make the comparison fair, the SMDP Q learner follows the same set of instructions whenever available. This large speedup is due to the generalization achieved by the instruction model. The parameter ζ is the probability of receiving an instruction at any decision point. Thus a larger ζ implies more number of instructions overall and also faster convergence. This is seen

[1]List of parameters used : $\epsilon = 0.07$, $\alpha = 0.1$, $k = 3$, $CThresh = 50$, $QThresh = 0.5$

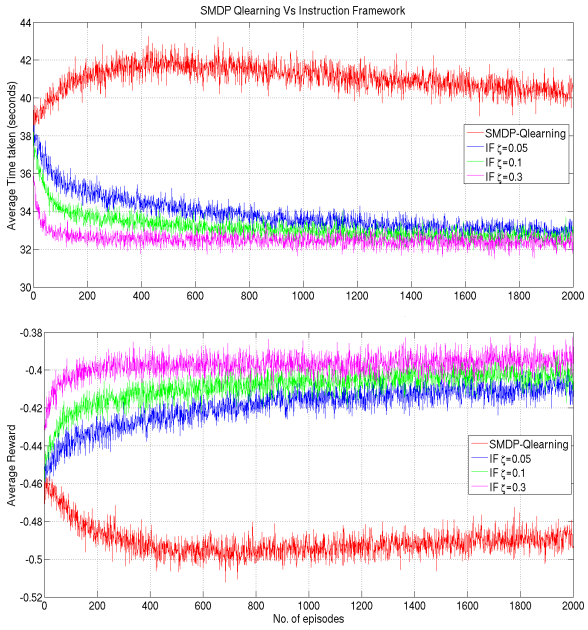


Figure 2: Comparison of SMDP Q learning with the instruction framework for various ζ

clearly in Fig 2. Approaches that employ function approximators (FAs) for the value function, proved to be difficult to setup for this domain. Using FAs such as a neural network and tile coding on this domain resulted in much longer learning periods than standard Q learning. Choosing options purely based on the k-NN classifier results in poorer performance than the above approach. This points out that our method is better than both generalizations of the policy and generalizations of the value function.

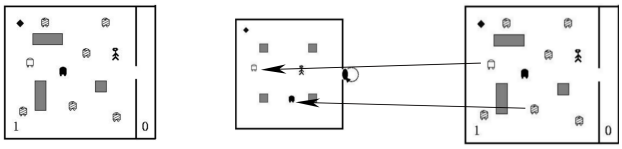
Game Domain[2]

This domain is well suited to showcase the advantages in using Φ-instructions. Solving this task requires the agent to carefully choose relevant features of the state space, which is one of the motivations for designing Φ-instructions. The agent uses these instructions to learn a pattern, if it exists, in this selection of features and exploits this knowledge to solve the task more efficiently.

The layout of the game is shown in Fig 3a. The environment has the usual stochastic gridworld dynamics and the *SLIP* parameter accounts for noise. The RL agent's goal is to collect the only diamond in the one room of the world and exit it. The agent collects a diamond by occupying the same square as the diamond. Possession of the diamond is indicated by a boolean variable, *have*.

The room is also populated by 8 autonomous adversaries. They are of three types - benign, delayer or retriever. Of the 8, only one is a delayer and another one is a retriever, the other 6 are benign. If the RL agent occupies the same square as the delayer it is considered captured and is prevented from making a move for a random number of time

[2]Some of the results for this domain have been reported elsewhere.



(a) (b)

Figure 3: (a) The Game domain. (b) Projections from the game world to the option world.

steps determined by a geometric distribution with parameter *HOLD*. When in a different square from the agent, the delayer pursues the agent with probability *CHASE*. The benign adversaries execute random walks and behave as mobile obstacles. The retriever behaves like the benign adversary as long as the diamond is not picked by the agent. Once the agent picks up the diamond, the retriever behaves like a delayer. The important difference is that once the retriever and the agent occupy the same square, the diamond is returned to its original position and the retriever reverts to being benign. The retriever also returns to being benign if the delayer has "captured" the agent. None of the adversaries can leave the room, and hence it is possible for the agent to "escape" from the room by exiting to the corridor. The agent is not aware of the types of the individual adversaries, nor is it aware of their *CHASE* and *HOLD* parameters. In every episode, a new pair of adversaries are chosen by the environment as the delayer and retriever. The RL agent can observe its own coordinates, the 8 adversaries' coordinates and the *have* variable.

Instruction Framework

Among the eight adversaries in the gameworld, only one is the delayer and one is the retriever. It is enough for the agent to observe these two adversaries to retrieve the diamond effectively. In other words, there are state features that can be ignored. Hence, we make projections of the states onto a subset of features resulting in a reduced world. A state in the game world is given by $s = \langle have, (x,y)_{agent}, (x,y)_{adv1}, \ldots, (x,y)_{adv8} \rangle$. The required state is given by $s_o = \langle have, (x,y)_{agent}, (x,y)_{del}, (x,y)_{ret} \rangle$. The projections used here are given by $\prod_{i \in D'} f_i(s)$, where $f_i : S \rightarrow S_i$ and D' is the reduced feature set. The agent uses Q-learning to learn the optimal policy on the reduced world.

The delayer and retriever change every episode and hence the corresponding projections also change. The agent does not know the true delayer and retriever. Φ type instructions are applicable here. We occasionally use these instructions to inform the agent about the indices of the adversaries that are the true delayer and true retriever. Suppose adv_k is the true delayer and adv_l is the true retriever for the current episode. The instruction $\Phi(s)$ gives the agent the $D' = \{have, (x,y)_{agent}, (x,y)_{adv_k}, (x,y)_{adv_l}\}$. When instructions are absent, the agent learns the correct (k,l) using a Bayesian weight update given by Ravindran et al. (2003).

Bayesian Weight Update Consider the set of cross product functions f characterized by $D' = \{have, (x,y)_{agent}, (x,y)_{adv_i}, (x,y)_{adv_j}\}$. There are 8 possibilities for both adv_i and adv_j resulting in a set of 64 cross product functions f^m. The likelihood of any f^m being the required cross product function is maintained using a factored weight vector $\langle w_n^1(\cdot,\cdot), w_n^2(\cdot,\cdot) \rangle$, with one component each for the delayer and retriever. The retriever component captures the dependence of the retriever on the delayer.

$$w_n^l(f^m, \psi(s)) = \frac{\overline{P^l}((\rho_{J_m}(s), a, \rho_{J_m}(s')).w_{n-1}^l(f^m, \psi(s))}{\mathcal{K}} \quad (5)$$

where $\psi(s)$ is a function of s that captures the features of the states necessary to distinguish the particular sub-problem under consideration, s' is the next state in the gameworld, J_i is the corresponding subset of features to be used for projecting onto the reduced MDP. $\overline{P^l}(s,a,s') = max(\nu, P^l(s,a,s'))$. $\mathcal{K}$ is the normalizing factor. $P^l(s,a,s')$ is the "projection" of $P(s,a,s')$ onto the subset of features J_m required in the computation of $w_n^l(f^m, \psi(s))$. For details about the "projection", refer to (Ravindran, Barto, and Mathew 2007).

Exploiting Instructions

In this section, we report additional experiments in which the agent exploits instructions in estimating *CHASE* and *HOLD* of the adversaries. The agent identifies the delayer and retriever assignment pattern in the environment based on these parameters. It uses this to reduce the number of updates required to identify the true delayer and retriever in the absence of instructions.

When the correct delayer-retriever pair is given as an instruction, the agent estimates the *CHASE* of the adversary that is the true delayer for the current episode. Similarly it can estimate the delayer's *HOLD* too. After a few such instructions, it would have good estimates of every adversaries' parameters. In order to show that this additional knowledge can be exploited effectively, we modify the game such that, for a given episode, only an adversary with *CHASE* ≤ 0.7 is chosen by the environment to be the delayer. A retriever is chosen from those adversaries with *CHASE* more than 0.7. A classifier is used to learn this model as more and more instructions are received. Possible retrievers and delayers are predicted using this model. This prediction is used to reduce the number of updates the agent needs to perform. For example, let the *CHASE* of the adversaries be $\{0.5, 0.9, 0.6, 0.7, 0.4, 0.35, 0.8, 0.73\}$. An episode's delayer will be selected only from adversaries 1, 3, 4, 5 and 6 and the retriever from the rest. Once this classification is learnt, we can avoid updating the weights for $\{1,3,4,5,6\}$ while learning the correct retriever and $\{2,7,8\}$ while learning the correct delayer. Overall, the agent converges to the true likelihoods in lesser updates as shown for the delayer in Fig 4d. S

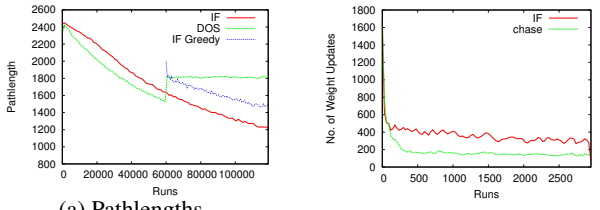
Analysis

We compare the performance of our approach (labeled IF) with the DOS approach proposed by Ravindran et al. (2007). In the DOS approach, the RL agent prelearns an optimal policy in a reduced MDP (training MDP) with feature set $D' = \{have, (x,y)_{agent}, (x,y)_{adv_{del}}, (x,y)_{adv_{ret}}\}$. This is

shown in Fig 3b. It lifts this optimal policy onto the game-world MDP by choosing f^m according to the Bayesian weight updates. In this approach, f_i is $\rho_{D'}(s)$, where the projections are onto the option MDP. Learning takes place only in the training phase.

Estimating *CHASE* and *HOLD* parameters using the DOS approach is not straight forward. The likelihood estimates for the delayer and retriever fluctuate heavily initially and we do not have a bound on when they converge. Thus the agent would not know when to start estimating the *CHASE* of the delayer.

DOS (Ravindran, Barto, and Mathew 2007) trains in the option MDP over 60000 episodes. IF does not have an exclusive learning phase, instead, depending on the availability of instructions, it alternates between learning using instructions and learning using the weight update. Hence all the weight updates shown for DOS occur after the learning phase. The graphs comparing the performance of Instruction Framework (IF) and DOS are shown in Fig 4.



(a) Pathlengths (b) Exploiting Instructions

Figure 4: Graphs comparing IF and DOS. The graphs have been Beizer smoothed only for visibility purposes. The trends in data are evident even in the non-smoothed plots. All results have been averaged over 50 independent runs.

Fig 4a shows the no. of time steps taken by each algorithm to solve the task successfully (collect the diamond and exit the room). It can be noticed that even though DOS performs well during the training phase, its performance drops in the game world. This is because of the differences in the training world and the game world in terms of obstacles, *CHASE* and *HOLD* parameters of adversaries. A major factor is the time taken to identify the true delayer and true retriever, until which the agent has an incomplete understanding of the game world. The figure suggests that IF does not face this problem. Even though IF does not suffer from losses due to differences in worlds, it is affected by time taken to identify true delayer and retriever. This is masked in the plot as the episodes have been averaged over independent runs during which the same episode would have received instructions in some runs and would not have in other runs. In order to show that IF outperforms DOS, *IFgreedy* has been plotted.

At regular intervals, the IF algorithm was made to imitate DOS in the sense that IF behaved greedily based on the knowledge of the game world it had at that moment. In addition to this, there were no instructions available to IF during these episodes resulting in time being spent identifying the true delayer and the true retriever. It can be seen in the figure that even IFgreedy performs better than DOS proving that IF outperforms DOS.

The no. of weight updates required by IF to identify the true delayer and retriever are comparable to DOS. In Fig 4b, the no. of weight updates required to identify the true delayer using the classifier based on *CHASE* (plotted as chase) and without (plotted as IF) are shown. *chase* does not consider those adversaries classified as retrievers in identifying the delayer. Hence instead of updating the likelihood of 8 adversaries, it only updates 5. This implies lesser no. of weights to update and hence an earlier convergence to the true delayer. It can be seen that the no. of updates required is nearly half that required when we do not make use of *CHASE* values based classification.

Conclusion and Future work

We have proposed a new framework to integrate instructions and showed with experimental validation that our approaches perform better in comparison to evaluation based learning methods. We have also discussed two types of instructions with mathematical formulations. Our framework is able to exploit the structural regularities in the domain either through indirect specifications ($\pi - instructions$ - Transporter domain) or direct specifications ($\Phi - instructions$ - Game domain). Also, we have shown that our approach performs better than other standard approaches.

Note that the representation of the policy in the Game domain is implicit via the Q function and hence the Φ-instructions are in effect used to derive an abstract representation for the value function.

One direction of extending our work, would be to consider instructions that translate to more complicated operations on the state space. Another interesting direction of future work would be to generalize over both instructions and value functions. It would be interesting to look at combining Φ-instructions and π-instructions.

References

Atkeson, C. G., and Schaal, S. 1997. Robot learning from demonstration. ICML '97, 12–20. Morgan Kaufmann Publishers Inc.

Barto, A. G., and Mahadevan, S. 2003. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems* 13:341–379.

Chapman, D. 1991. *Vision, instruction, and action*. MIT Press.

Clouse, J. A., and Utgoff, P. E. 1992. A teaching method for reinforcement learning. ICML92, 92–101. Morgan Kaufmann

Maclin, R., and Shavlik, J. W. 1998. Creating advice-taking reinforcement learners. In S.Thrun, and L.Pratt., eds., *Learning to Learn*, 22, 251–281. Kluwer Academic Publishers.

Ng, A. Y., and Russell, S. 2000. Algorithms for inverse reinforcement learning. ICML00, 663–670. Morgan Kaufmann.

Price, B., and Boutilier, C. 2003. Accelerating reinforcement learning through implicit imitation. *JAIR* 19:569–629.

Ravindran, B.; Barto, A. G.; and Mathew, V. 2007. Deictic option schemas. IJCAI07, 1023–1028. AAAI Press.

Rosenstein, M. T., and Barto, A. G. 2004. Supervised actor-critic reinforcement learning. In *Learning and Approximate Dynamic Programming: Scaling Up to the Real World*, 359–380. John Wiley and Sons.

Zeigler, B. P. 1972. Toward a formal theory of modeling and simulation: Structure preserving morphisms. *J. ACM* 19:742–764.