Between Imitation and Intention Learning

James MacGlashan
Brown University
james_macglashan@brown.edu

Michael L. Littman
Brown University
mlittman@cs.brown.edu

Abstract

Research in learning from demonstration can generally be grouped into either *imitation learning* or *intention learning*. In imitation learning, the goal is to imitate the observed behavior of an expert and is typically achieved using supervised learning techniques. In intention learning, the goal is to learn the intention that motivated the expert's behavior and to use a planning algorithm to derive behavior. Imitation learning has the advantage of learning a direct mapping from states to actions, which bears a small computational cost. Intention learning has the advantage of behaving well in novel states, but may bear a large computational cost by relying on planning algorithms in complex tasks. In this work, we introduce *receding horizon inverse reinforcement learning*, in which the planning horizon induces a continuum between these two learning paradigms. We present empirical results on multiple domains that demonstrate that performing IRL with a small, but non-zero, receding planning horizon greatly decreases the computational cost of planning while maintaining superior generalization performance compared to imitation learning.

1 Introduction

In *Learning from Demonstration* (LfD), an agent derives a policy (a mapping from states to actions) for an environment by observing an expert's behavior. Two widely studied approaches to this problem include *imitation learning* and *intention learning*. Imitation learning learns a direct mapping from states to actions, typically with a supervised learning algorithm where the inputs are state features and the output is an action label. Intention learning is often framed as an inverse reinforcement learning (IRL) [Ng and Russell, 2000] problem in which the goal is to learn a reward function that would motivate the observed training behavior, and then use a planning algorithm to indirectly derive the policy that maximizes the learned reward function. In this way, intention learning can be characterized as an LfD algorithm that has a "decision-making bias" that leverages the environment's transition dynamics and state features, whereas immitation learning limits itself to state features.

Each paradigm has advantages. In imitation learning, the policy is learned directly, so the only computation required when the agent behaves in the world is evaluating the learned classifier. However, if the agent is in a novel state that is not sufficiently similar to states seen in the training data, behavior may suffer. Inversely, intention learning may have a high computational cost—both at training and test time—due to its reliance on a planning algorithm in the decision loop, but may generalize better to novel states since its planning algorithm can produce novel behavior that maximizes the expert's intentions. Depending on the richness of the task, the computational complexity may be prohibitive; in fact, a designer may prefer imitation learning precisely because solving the true task directly with a planning algorithm is too computationally demanding.

In this work, we introduce *receding horizon inverse reinforcement learning* (RHIRL), which defines an IRL problem of finding a reward function that causes a receding horizon controller (RHC) to match the expert's behavior. An RHC is a policy that, in each state, maximizes the expected future discounted reward up until some time interval in the future. In the context of IRL, an RHC has the important property that the horizon defines a continuum between imitation and intention learning. With a planning horizon of zero, RHIRL will seek a reward function that directly defines the preferences for actions in each state, thereby imitating the training data and requiring very low computational cost. With an infinite horizon, we recover the standard IRL problem with a potentially large computational cost for planning, depending on the richness of the task. By using intermediate planning horizon values, however, the computational complexity of the planning algorithm can be fixed independent of the task complexity, while also affording the agent some ability to look into the future to account for novel situations. In effect, RHIRL learns a representation of the task along with planning knowledge that guides behavior.

We present empirical results on three domains: a navigation domain, the classic reinforcement-learning mountain car domain [Singh and Sutton, 1996], and a lunar lander game [MacGlashan, 2013]. We show that even a small horizon can result in vastly superior test performance on novel states compared to imitation learning. We have also made

RHIRL publically available as part of BURLAP[1], an open source reinforcement learning and planning library.

2 Markov Decision Processes

Like other IRL algorithms, we formalize the decision-making problem an agent faces as a *Markov Decision Process* (MDP). An MDP is defined by the tuple (S, A, T, R), where S is the set of states in the world; A is the set of actions that the agent can take; $T(s'|s,a)$ defines the transition dynamics: the probability of the agent transitioning to state $s' \in S$ after taking action $a \in A$ in state $s \in S$; and $R(s,a,s')$ is the reward function, which returns the reward the agent receives for taking action a in state s and transitioning to state s'.

The goal of planning or learning in an MDP is to find a policy $\pi : S \rightarrow A$ (a mapping from states to actions) that maximizes the expected future discounted reward: $E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \mid \pi\right]$, where $\gamma \in [0,1]$ is a discount factor specifying how much immediate rewards are favored over distant rewards, and s_t and a_t are the state and action taken at time t, respectively.

Two important concepts for planning and learning are the *state value function* (V) and the *state-action value function* (Q). The optimal state value function (V^*) specifies the expected future discounted reward when following the optimal policy for each state; the Bellman equation defines this function recursively as

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s'|s,a) \left[R(s,a,s') + \gamma V^*(s')\right]. \quad (1)$$

The optimal state-action value function (Q^*) specifies the expected future discounted reward when taking each action in each state and then following the optimal policy thereafter. It is similarly recursively defined as

$$Q^*(s,a) = \sum_{s' \in S} T(s'|s,a) \left[R(s,a,s') + \gamma V^*(s')\right]. \quad (2)$$

Note that the state value function may be written in terms of the state-action value function:

$$V^*(s) = \max_{a \in A} Q^*(s,a). \quad (3)$$

Many planning and learning algorithms estimate the optimal state value function or state-action value function directly and then set the policy to $\pi(s) = \max_{a \in A} Q(s,a)$ [Bertsekas, 1987].

When computing a value function is too challenging, approximations can be used instead. One common form of approximation is *receding horizon control* (RHC). In RHC, the agent selects actions for each state according to a *finite horizon value function* for that state. A finite horizon value function defines the expected future discounted reward for h steps into the future (the horizon). The value functions for a horizon h are recursively defined as

$$V^h(s) = \max_{a \in A} \sum_{s' \in S} T(s'|s,a) \left[R(s,a,s') + \gamma V^{h-1}(s')\right] \quad (4)$$

and

$$Q^h(s,a) = \sum_{s' \in S} T(s'|s,a) \left[R(s,a,s') + \gamma V^{h-1}(s')\right], \quad (5)$$

where $V^0(s) = 0$. Similarly to Equation 3, the finite horizon state value function can be rewritten as the maximum finite horizon Q-value. The optimal policy for an RHC with a horizon of h is defined as $\pi^h(s) = \max_{a \in A} Q^h(s,a)$. The name *receding horizon* refers to the fact that, after each decision, the planning horizon recedes one more step from where it was when the previous decision was made.

3 Learning from Demonstration

In *Learning from Demonstration* (LfD), the goal is to take as input example behavior from an expert acting in an MDP and learn a policy that can replicate the expert's behavior and performance in the environment. In some contexts, the reward function the expert is trying to maximize is assumed to be known and can be leveraged to improve performance [Abbeel and Ng, 2005; Walsh *et al.*, 2010]. Here, we make the more standard IRL assumption that we do not know the expert's underlying reward function, but instead have knowledge of the MDP's states, actions, and transition dynamics and examples of expert behavior. Ideally, the learned policy should generalize so that it can replicate the expert's behavior even in states unobserved in the demonstrations. We formalize training demonstrations as a set of MDP *trajectories*. An MDP trajectory (t) is a sequence of $|t| = n$ state-action pairs, $t = \langle (s_1, a_1), ..., (s_n, a_n) \rangle$, observed from the expert starting in some initial state of the environment (s_1) and acting for n time steps. Therefore, each state s_i in the trajectory is expected to be drawn according to $s_i \sim T(s_i | s_{i-1}, a_{i-1})$, except for the first state (s_1), which is assumed to be drawn from some other unknown initial state distribution or selected by the expert. For goal-directed terminating tasks, trajectories end when a goal state is reached.

We next describe two approaches to LfD: imitation learning and intention learning.

3.1 Imitation Learning

We define imitation learning as an LfD problem that only has access to state features and the action labels in the trajectory. This problem can be formulated as a supervised learning classification problem in which the inputs (x) are state features and the output label (y) is the action to take. To create the supervised learning dataset, we assume access to a state-feature vector function $\phi : S \rightarrow \mathbb{R}^m$. Given a set of expert trajectories D, the training data input X is defined as the state-feature vector for each state in each state-action pair of each trajectory in the dataset: $X = \{\phi(s) \mid (s,a) \in \bigcup_{t \in D} t\}$; the training data output Y is defined as the set of actions in each state-action pair of each trajectory in the dataset: $Y = \{a \mid (s,a) \in \bigcup_{t \in D} t\}$.

Once a supervised learning dataset has been created, any "off-the-shelf" classification algorithm can be used to learn a policy. In the past, various classification algorithms, such as decision trees and support vector machines, have been used [Pomerleau, 1993].

[1]http://burlap.cs.brown.edu/

3.2 Intention Learning

In contrast to imitation learning, we define intention learning as a LfD problem that in addition to the state features, also uses the transition dynamics of the environment and seeks a goal that motivates the observed expert behavior. In the context of MDPs, intention learning can be framed as an inverse reinforcement learning (IRL) problem, where the the intention of the expert is captured by the reward function they are trying to maximize [Ng and Russell, 2000]. IRL solutions effectively generalize to novel states that require novel unobserved behavior, because even a simple reward function can induce complex behavior for many states. For example, many goal-directed MDP tasks have a simple reward function where every state returns a value of -1, except the goal, which returns some non-negative value [Koenig and Simmons, 1993].

Although there are other IRL formalizations, we adopt the formalization that treats finding a reward function that motivates the observed behavior as an inference problem [Ramachandran and Amir, 2007; Babes *et al.*, 2011; Lopes *et al.*, 2009; Ziebart *et al.*, 2008]. Specifically, given a dataset D of trajectories, we seek the maximum likelihood reward function R. Our approach follows Babeş *et al.* [2011], in which the likelihood of D given reward function R is defined as

$$L(D|R) = \prod_{t \in D} \prod_{i}^{|t|} \pi_R(s_i, a_i), \tag{6}$$

where $\pi_R(s,a)$ is a stochastic policy defining the probability of taking action a in state s when the reward function to be maximized is R. Specifically, we use the Boltzmann (softmax) policy

$$\pi(s,a) = \frac{e^{\beta Q(s,a)}}{\sum_{a' \in A} e^{\beta Q(s,a')}}, \tag{7}$$

where β is a parameter that controls how noisy the policy is with respect to the action with the maximum Q-value. As β approaches infinity, the softmax policy deterministically selects actions with the maximum Q-value; when $\beta = 0$, the softmax policy selects actions uniformly at random. When the expert's behavior is expected to be noisy and suboptimal, it is useful to set β nearer to zero. (The β parameter can be optimized automatically as well, but we do not take that approach here.)

To quickly find the maximum likelihood reward function, Babeş *et al.* assume that the reward function is differentiable and parameterized by a vector θ. Then, gradient ascent in the log likelihood of the parameter space is used to find the maximum likelihood reward function. The gradient of the log likelihood is

$$\nabla \log\left[L(D|\theta)\right] = \sum_{t \in D} \sum_{i}^{|t|} \frac{\nabla \pi_\theta(s_i, a_i)}{\pi_\theta(s_i, a_i)}, \tag{8}$$

where

$$\nabla_\theta \pi(s,a) = \frac{\beta \nabla_\theta Q(s,a) Z_a Z - Z_a \left[\sum_{a'} \beta \nabla_\theta Q(s,a') Z_{a'}\right]}{Z^2}, \tag{9}$$

$Z_a = e^{\beta Q(s,a)}$, and $Z = \sum_{a' \in A} Z_{a'}$.

Note that computing the gradient of the softmax policy requires computing the gradient of the Q-function, which typically isn't differentiable everywhere due to the max operator in the value function. Babeş *et al.* resolve this difficulty by changing the max operator in the value function in Equation 3 to be a Q-value weighted softmax distribution $\hat{V}(s) = \sum_{a \in A} Q(s,a)\pi(s,a)$. This value function is differentiable everywhere and as β approaches infinity, it converges to the standard state-value function.

4 Receding Horizon IRL

We now introduce *Receding Horizon IRL* (RHIRL). In RHIRL, the goal is to find a reward function that motivates an RHC to match the observed expert behavior. Adopting the same approach as before, this search is formalized as finding the maximum likelihood reward function where the stochastic policy is a softmax variant of the receding horizon controller. Therefore, for a horizon of h, the likelihood function in Equation 6 is modified to use a softmax policy of the finite horizon Q-values ($\pi^h(s,a)$).

For many standard planning tasks, RHCs may require a very large horizon to well approximate the optimal infinite horizon policy. For example, goal-directed tasks typically have very sparse rewards, only valuing reaching a distant goal state. For these reward functions, RHCs behave randomly until the goal state is within the horizon and the RHC can hone in on it. However, when RHIRL searches over a sufficiently expressive reward function space, it need not suffer this same limitation, because it can include local "shaping" rewards as well as the rewards needed to represent the task [Ng *et al.*, 1999].

For example, suppose the expert is in actuality trying to maximize an infinite horizon task, but the RHIRL horizon is set to 1. In this case, a valid 1-step horizon reward function that can reproduce the same behavior is a reward function that is structurally the same as the state value function for the infinite horizon task. Naturally, a horizon of 1 provides the agent limited ability to generalize to novel states, but, as the horizon grows, the agent can react better to task specific features. With a sufficiently large horizon, the best RHIRL reward function may in fact be the actual reward function the expert was trying to maximize. We explore this effect further in our navigation task results (Section 5.1).

To efficiently find an RHC reward function that matches the expert's behavior, we adapt the softmax value function and gradient ascent approach employed by Babeş *et al.* to finite horizon planning. That is, we assume the reward function is differentiable and parameterized by a vector θ and compute the softmax value function, and its gradient, using a recursive tree-search algorithm as shown in Algorithm 1.

The algorithm takes as input the state (s) and planning horizon (h). When the horizon is zero, the value function and gradient is set to zero. For all other cases, the algorithm first recursively computes the value function and gradient for all possible transition states for $h - 1$. When the number of possible transitions from a state is large or infinite, the transition function in the tree-search algorithm can be replaced

Algorithm 1 ComputeValue(s, h)

```
if $h = 0$ then                                   $\triangleright$ recursive base case
    $\hat{V}^h(s) := 0$
    $\nabla_\theta \hat{V}^h(s) := 0$
    return
end if
for $a \in A$ do
    $S' := \{s' \in S \mid T(s'|s,a) > 0\}$
    for $s' \in S'$ do
        computeValue($s'$, $h-1$)
    end for
    $\hat{Q}^h(s,a) := \sum_{s' \in S'} T(s'|s,a) \left[R(s,a,s') + \gamma \hat{V}^{h-1}(s')\right]$
    $\nabla_\theta \hat{Q}^h(s,a) := \sum_{s' \in S'} T(s'|s,a) \times \left[\nabla_\theta R(s,a,s') + \gamma \nabla_\theta \hat{V}^{h-1}(s')\right]$
end for
$\hat{V}^h(s) := \sum_{a \in A} \hat{Q}^h(s,a) \pi_\theta^h(s,a)$
$\nabla_\theta \hat{V}^h(s) := \sum_{a \in A} \nabla_\theta \hat{Q}^h(s,a) \pi_\theta^h(s,a) + \hat{Q}^h(s,a) \nabla \pi_\theta^h(s,a)$
```

with a sparse sampled approximation [Kearns *et al.*, 1999] that maintains a bounded computational complexity. Next, the algorithm computes the Q-value and Q-value gradient, which are functions of the next states' value and value gradient for horizon $h-1$, which was recursively computed in the previous step. Using the Q-values and Q-value gradients, the state value function and value function gradient is set for the current state and horizon. This algorithm can be trivially memoized to prevent repeated computation of the same state-horizon value function.

Given the finite horizon value function and its gradient, the gradient of the receding horizon policy is trivially computed and gradient ascent is used to search for the maximum likelihood reward function.

Under our normal definition for a finite horizon value function, the state value at $h = 0$ is zero. This definition would cause an RHC with a horizon of zero to behave randomly. However, it is more useful to think of an RHC controller with $h = 0$ as the case when the agent cannot look any steps into the future; as an imitation learning problem. Instead then, we treat an RHC with $h = 0$ as a special case that uses an instantaneous reward function ($R(s,a)$) to encode action preferences. If a reward function is normally defined in terms of state features, it is trivial to turn it into state-action features that encode action preferences by maintaining a duplicate set of state features for each action. Similar to how an RHC horizon of 1 may result in learning a reward function that is similar to the value function of the infinite horizon task, a horizon of zero may result in learning a reward function that is similar to the Q-function of the infinite horizon task.

5 Experimental Results

We compare RHIRL's performance with various horizons to imitation learning on three domains: a navigation domain, the classic RL mountain car domain [Singh and Sutton, 1996], and a lunar lander game [MacGlashan, 2013]. For imitation learning, we use Weka's J48 classifier [Hall *et al.*, 2009], which is an implementation of the decision tree algorithm C4.5 [Quinlan, 1993], and Weka's logistic regression implementation. We found that J48 was superior to logistic regression in all cases except lunar lander. Therefore, we only report logistic regression's results in lunar lander. Because our goal is to test generalization performance, in all experiments we use very few demonstrations and then test the performance of the learned results on start states not observed in the demonstrations. We also report the total training CPU time. For RHIRL, training time includes the "replanning" necessary for all steps of gradient ascent. Note that for the continuous state domains (mountain car and lunar lander), exact solutions for an infinite horizon are intractable; for typical reinforcment learning in these kinds of domains, approximation methods, like value function approximation, using many thousands of samples are necessary and can require its own parameter tuning, making it challenging to solve the standard IRL problem in such domains.

To facilitate generalization, the learned reward function is a linear combination of both task features and *agent-space features* [Konidaris and Barto, 2006]. Agent-space features are features relative to the agent that have been used in the past to successfully learn reward shaping values. For the supervised learning baselines, we tested performance using both the state variables of the domain, and agent-space features (and others); however, the basline performance was always better using the domain's state variables, so we only report those results.

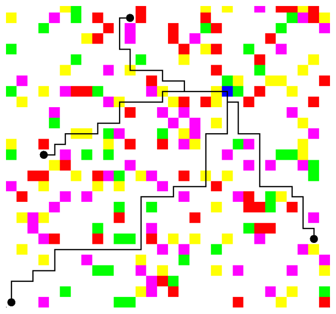
Figure 1: The four expert trajectories and cell type distribution in the 30x30 navigation task.

5.1 Navigation

The navigation task we tested is a 30x30 grid world in which the agent can move north, south, east, or west. Each cell in the grid world can either be empty or belong to one of five different cell types, each represented by a distinct color.

Expert demonstrations were optimal trajectories for a reward function that assigned a reward of zero for empty cells and yellow cells, a reward of one to blue cells, and a reward of -10 for the red, green, and purple cells. The expert trajectories and distribution of colored cells are shown in Figure 1. RHIRL used 10 steps of gradient ascent.

We compare the performance of RHIRL with horizons lengths 45, 6, 1, and 0 to the J48 supervised learning algorithm. The horizon of length 45 is used as an upperbound to represent the standard IRL case when the horizon is sufficiently large to find the goal location. In this case, we take

Algorithm	Start 1	Start 2	Goal	Training CPU
J48	1	4	F	0.27s
RHIRL $h = 0$	F	F	F	0.40s
RHIRL $h = 1$	2	3	2	0.83s
RHIRL $h = 6$	2	1	1	1.68s
RHIRL $h = 45$	**0**	**0**	**0**	8.0s

Table 1: Navigation performance and total training CPU time. Performance indicates the number of obstacles hit on the path to the goal, or if the agent failed to reach the goal entirely (indicated by an "F").

a more standard IRL approach of setting the reward function features to only task features: five binary features that indicate whether the agent is in each type of cell. For the shorter horizons, the reward function features include the same task features and five agent-space features that specify the Manhattan distance of the agent to the nearest cell of each type, for a total of ten features.

After training, we consider three test cases: two in which the start state of the agent is novel from any of the demonstrations ("Start 1" and "Start 2") and a third in which the start state of the agent has been previously observed, but the goal blue cell is somewhere new ("goal"). Table 1 shows the performance for each test. An F indicates that the agent failed to reach the goal cell; otherwise, the number indicates the number of obstacles the agent hit along the path to the goal (smaller is better).

Imitation learning generally performed poorly. After learning, J48 only managed to solve the two tasks with different start states, and RHIRL with $h = 0$ could not solve any of the tasks. However, when $h > 0$, RHIRL was able to successfully solve all tasks. With larger horizons, the agent was better at avoiding obstacles.

To illustrate how RHIRL learns not just a description of the task, but reward shaping values that guide the agent, Figure 2 projects onto the grid world the objective reward function used to generate the expert trajectories, the state value function under the objective reward function, and the learned reward function under RHIRL with $h = 6$. Note that the objective reward function is very discontinuous, whereas its value function is a fairly smooth gradient toward the goal. Rather than learn the discontinuous objective reward function, RHIRL's learned reward function captures a similar gradient that guides the agent in the correct direction despite its small horizon.

5.2 Mountain Car

In the mountain car domain, the agent is tasked with driving an under powered car out of a valley. The agent has three actions: accelerate forward, accelerate backwards, and coast. To get out of the valley, the agent needs to rock back and forth to gain momentum. The problem is described by two state variables: the car's position in the valley and its velocity. The task is complete when the agent reaches the top of the front side of the valley.

For learning, we use a set of nine radial basis functions dis-

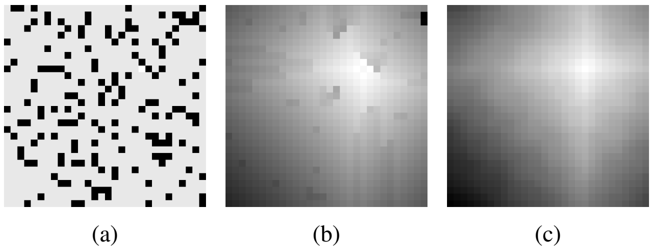

Figure 2: The navigation task objective reward function (a), objective value function (b), and RHIRL ($h = 6$) learned reward function (c). Whiter spots indicate larger values. Note that the learned reward function resembles the objective value function more than the objective reward function.

Algorithm	Good Demo.	Bad Demo.	Training CPU
J48	115	F	0.32s/0.23s
RHIRL $h = 0$	**107**	F	0.47s/0.32s
RHIRL $h = 1$	134	147	0.86s/0.74s
RHIRL $h = 4$	139	**141**	6.80s/4.01s

Table 2: Mountain car performance and total training CPU time. Performance is measured in the number of steps to exit the valley. An "F" indicates that the agent failed to complete the task. CPU time for each demonstration is reported.

tributed over the state space, which is a feature set often used for value function approximation in reinforcement learning for this task. We evaluate learned performance from the bottom of the valley under two different training conditions. In the first training condition, the agent must learn from a single trajectory that starts halfway up the front side of the valley, backs down to the opposite side of the valley, and then powers forward to escape. In the second training condition, the agent must learn from a single trajectory that shows the agent starting at the top of the back side of the valley and powering forward the entire way to the top of the font side of the valley. In both conditions, the demonstrations were provided by an author of the paper and may be suboptimal with respect to the task. We consider the demonstration in the first condition a "good" demonstration because it exhibits the rocking behavior necessary to succeed when stating in the valley; we consider the second demonstration a "bad" demonstration since it does not demonstrate the critical rocking behavior. RHIRL used 15 steps of gradient ascent.

As before, we compared RHIRL with various horizons against J48. Table 2 shows the number of steps taken to complete the task for each algorithm under each training condition. For the good demonstration, imitation learning with RHIRL ($h = 0$) performs the best. The decrease in performance as the horizon increases is possibly due to the fact that, in the demonstration, the expert overshot going backwards by failing to begin slowing down soon enough. With increased ability to look ahead, RHIRL may try to match this observed trajectory. In contrast, imitation learning will observe the forward acceleration actions on the back side of the valley with

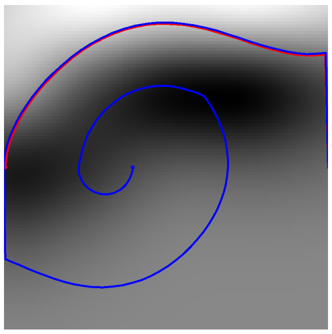

Figure 3: The RHIRL ($h = 1$) learned reward function for the mountain car task. The x-axis is the position of the car; the y-axis its velocity. The red curve the is "bad" demonstration and the blue curve is the learned behavior starting in the valley.

a high velocity and match those overall preferences, thereby encouraging the agent to reverse sooner.

For the bad demonstration, both imitation learning variants fail. Remarkably, however, RHIRL with a horizon as small as 1 is able to complete the task. To investigate why, Figure 3 projects the $h = 1$ learned reward function onto a plot of the position and velocity variables. The figure also projects the expert demonstration (in red) and the trajectory of the learned agent (in blue). The results show that at slow or negative velocities, the agent prefers the back valley since the demonstration is at its slowest there. As a consequence, the agent is first motivated to back up, and then power forward once it can gain forward momentum. However, this momentum is not enough to climb the front valley completely, so once it loses its forward momentum, it is again motivated to go to the back of the valley. At this point, the agent now is in a similar position as the start of the demonstration and it powers forward.

5.3 Lunar Lander

The lunar lander domain is a simplified version of the classic Atari game by the same name. In this domain, an agent pilots a lunar lander and is tasked with taking off and landing on a landing pad. The agent has five possible actions: a powerful rocket thrust, a weak rocket thrust, coast, and rotate clockwise and counterclockwise. The state variables defining the task include the agent's x and y position, angle of rotation, and x and y velocities.

For learning, 12 state features were used. Two features were binary; one for indicating that the agent was touching the ground and another that it had landed on the landing pad. Nine features were radial basis functions that depended on the x and y position of the agent (not its rotation or velocity) and were spaced uniformly across the area. A final radial basis function was placed on the top of the landing pad.

The training data consisted of two expert trajectories provided by an author of the paper, which could be suboptimal; one that started very near the landing pad, and another a medium distance from it. Both training trajectories are shown in Figure 4. Test performance from two initial states was evaluated; first from an "easy" start position that was between the

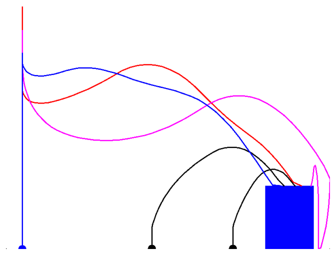

Figure 4: The Lunar Lander task. The blue box is the landing pad. The black lines represent the expert trajectories used for learning. The red, purple, and blue lines represent the RHIRL learned solutions for horizons of one, three, and five, respectively.

start position of the two demonstrations, and a "hard" start position that was much further from the landing pad than either training trajectory. RHIRL used 10 steps of gradient ascent.

Table 3 shows the performance in terms of number of steps for the agent to complete the task in each test condition. Logistic regression does well on the easy start, but fails to generalize to the hader start. Moreover, reviewing the resulting trajectory for the easy start reveals that the agent nearly perfectly replicates one of the expert trajectories, only rotating one step earlier and using one strong thrust in place of a weak trust, suggesting that the agent only performed well because memorizing one of the trajectories is sufficient to behave well in the easy start.

In contrast, RHIRL with a small lookahead is able to successfully pilot the ship to the landing pad for both easy and hard starts. In particular, there is a large improvement in performance for $h = 5$. This increase in performance is likely due to the fact that once the agent can look ahead far enough, it can more stably determine how to pilot the ship closer to the landing pad (which can require steps of rotating and thrusting). This effect is illustrated in Figure 4, which shows the learned trajectories for RHIRL with $h = 1$, $h = 3$, and $h = 5$ from the hard start position. In all cases, the agent first launches into the air, having learned from the demonstrations that the ground is to be avoided. However, only for $h = 5$ does the agent quickly move toward the landing pad after taking off; learning with smaller horizons causes the agent to tend to spend a good deal of time trying to stablize vertical velocity before finding its way toward the landing pad.

6 Conclusion

In this work, we introduced *Receding Horizon Inverse Reinforcement Learning* (RHIRL), which uses IRL techniques to learn a reward function for a receding horizon controller (RHC) from expert demonstrations. Although RHCs are often unable to behave well in normal planning tasks, IRL learns a reward function that shapes the agent's behavior in ways that compensate for the short lookahead. Furthermore, the horizon of RHIRL defines a continuum between imitation and intention learning where a horizon of zero instantiates an

Algorithm	Easy	Hard	Training CPU
Logistic Regression	**34**	F	0.38s
RHIRL $h = 0$	F	F	1.41s
RHIRL $h = 1$	184	242	1.67s
RHIRL $h = 3$	188	115	4.15s
RHIRL $h = 5$	57	**72**	29.50s

Table 3: Lunar Lander performance and total training CPU time. Performance is the number of steps to land on the landing pad. An "F" indicates that the agent failed to complete the task.

imitation learning problem in which action preferences are directly modeled, and a large horizon recovers the standard IRL problem.

We showed on multiple domains that even with very few demonstrations and a small horizon, RHIRL is able to successfully generalize to novel states, whereas traditional imitation learning methods can completely fail. As a consequence, RHIRL can capture much of IRL's ability to generalize without paying the large computational cost necessary for infinite horizon planning in complex domains.

In this work, we adapted the gradient ascent approach of IRL developed by Babeş *et al.* [2011]. This approach allows good reward functions to be quickly discovered. However, other IRL techniques might also be effective for RHIRL.

A limitation of RHIRL is that, for a non-zero horizon, access to the transition function is required. In the future, it would be useful to explore how well RHIRL works when only an approximate, or learned, transition function is available. If the learned reward function includes state-action features, it may be possible for RHIRL to learn rewards that help mitigate the effects of errors in the model.

References

[Abbeel and Ng, 2005] Pieter Abbeel and Andrew Y. Ng. Exploration and apprenticeship learning in reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*, pages 1–8, 2005.

[Babes *et al.*, 2011] Monica Babes, Vukosi N. Marivate, Michael L. Littman, and Kaushik Subramanian. Apprenticeship learning about multiple intentions. In *International Conference on Machine Learning*, pages 897–904, 2011.

[Bertsekas, 1987] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Hall *et al.*, 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[Kearns *et al.*, 1999] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1324–1331, 1999.

[Koenig and Simmons, 1993] Sven Koenig and Reid G. Simmons. Complexity analysis of real-time reinforcement learning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 99–105, Menlo Park, CA, 1993. AAAI Press/MIT Press.

[Konidaris and Barto, 2006] G.D. Konidaris and A.G. Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the Twenty Third International Conference on Machine Learning*, pages 489–496, June 2006.

[Lopes *et al.*, 2009] Manuel Lopes, Francisco Melo, and Luis Montesano. Active learning for reward estimation in inverse reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, pages 31–46. Springer, 2009.

[MacGlashan, 2013] James MacGlashan. *Multi-Source Option-Based Policy Transfer*. PhD thesis, University of Maryland, Baltimore County, 2013.

[Ng and Russell, 2000] Andrew Y. Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *International Conference on Machine Learning*, pages 663–670, 2000.

[Ng *et al.*, 1999] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287, 1999.

[Pomerleau, 1993] Dean A. Pomerleau. *Neural network perception for mobile robot guidance*. Kluwer Academic Publishing, 1993.

[Quinlan, 1993] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Ramachandran and Amir, 2007] Deepak Ramachandran and Eyal Amir. Bayesian inverse reinforcement learning. *International Joint Conference on Artificial Intelligence,*, pages 2586–2591, 2007.

[Singh and Sutton, 1996] Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1/2/3):123–158, 1996.

[Walsh *et al.*, 2010] Thomas J. Walsh, Kaushik Subramanian, Michael L. Littman, and Carlos Diuk. Generalizing apprenticeship learning across hypothesis classes. In *Proceedings of the Twenty-Seventh International Conference on Machine Learning (ICML-10)*, pages 1119–1126, 2010.

[Ziebart *et al.*, 2008] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, pages 1433–1438, 2008.