JMLR: Workshop and Conference Proceedings 20 (2011) 1–16 Asian Conference on Machine Learning

Improving Policy Gradient Estimates with Influence Information

Jervis Pinto PINTO@EECS.OREGONSTATE.EDU
Alan Fern AFERN.EECS.OREGONSTATE.EDU
Tim Bauer BAUERTIM@EECS.OREGONSTATE.EDU
Martin Erwig ERWIG@EECS.OREGONSTATE.EDU
School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331, USA.

Editor: Chun-Nan Hsu and Wee Sun Lee

Keywords: side information, policy gradient RL, adaptation-based programming

Abstract

In reinforcement learning (RL) it is often possible to obtain sound, but incomplete, information about influences and independencies among problem variables and rewards, even when an exact domain model is unknown. For example, such information can be computed based on a partial, qualitative domain model, or via domain-specific analysis techniques. While, intuitively, such information appears useful for RL, there are no algorithms that incorporate it in a sound way. In this work, we describe how to leverage such information for improving the estimation of policy gradients, which can be used to speedup gradient-based RL. We prove general conditions under which our estimator is unbiased and show that it will typically have reduced variance compared to standard unbiased gradient estimates. We evaluate the approach in the domain of Adaptation-Based Programming where RL is used to optimize the performance of programs and independence information can be computed via standard program analysis techniques. Incorporating independence information produces a large speedup in learning on a variety of adaptive programs.

1. Introduction

Standard off-the-shelf reinforcement learning (RL) algorithms often do not scale well for large, complex sequential decision making problems. This has led to research that extends RL algorithms to leverage various types of additional inputs, or *side information*, in order to accelerate learning. For example, such side information has included shaping rewards (Ng et al., 1999) and hierarchical task structures in the form of MAXQ hierarchies (Dietterich, 1998) or ALISP programs (Andre and Russell, 2000). While often effective, such information may require significant human expertise and only captures a subset of the potentially useful side information that might be available in a domain.

In this paper, we consider a form of side information that has not yet been exploited for RL, yet is often available at little to no cost to a designer. In particular, we consider side information in the form of assertions about context-specific influences and independencies among state variables, decisions, and rewards. Such assertions are often easily elicited from a domain expert or can be automatically computed given a partial, qualitative domain model. Further, in our motivating application of Adaptation-Based Programming (ABP), where RL is used to optimize the performance of programs, the independencies can be automatically extracted using standard program analysis techniques with no extra effort from the user (see Section 6). As a simpler example to help motivate this form of side information, consider the following illustrative RL problem, which will be used as a running example throughout the paper.

© 2011 J. Pinto, A. Fern, T. Bauer & M. Erwig.

Illustrative Example. Consider a robot moving in a grid world where certain grid cells have buttons that can be pressed to receive rewards. On even numbered time-steps, the robot may either move one step in any cardinal direction or "do-nothing", whereas at odd numbered time-steps the robot may either choose to execute a "press" action or "do-nothing". The state of the process is given by the robot's current location (X, Y), a binary variable B indicating whether there is a button at the current location, and a binary flag O which is 1 at odd time steps and 0 otherwise, indicating which type of action is applicable. The press action results in a fixed negative reward in a button-less cell, and a reward of $x + y$ in a cell (x, y) that has a button. In all other cases, zero reward is obtained. The objective is to maximize the sum of rewards over T steps.

Now suppose that reward r_t is obtained after a button press at an odd time step t. A traditional RL algorithm must solve the credit assignment problem of deciding which previous decisions had an "influence" on the value of r_t. However, it is straightforward from the above description to identify which decisions are provably irrelevant to the value of r_t. In particular, all previous "move" decisions can potentially change the location of the robot which affects the reward so all decisions at even-numbered time-steps are potentially relevant as well as the "push" decision at time t. However, the remaining decisions at prior odd time steps ("press" or "do-nothing") cannot influence the value of r_t. Intuitively, providing such influence information to an RL algorithm has the potential to significantly simplify the credit assignment problem and hence speedup learning. In the absence of such information, standard RL will need to learn through trial and error that button presses deserve no credit for rewards at later time steps. Unfortunately, no existing RL algorithms are able to leverage side information about such influences, or lack of influences, when available.

Contributions. Our main contributions in this work are: 1) A formalization of the above type of side information, 2) A modified policy gradient estimator that uses this information to speedup policy-gradient RL, 3) A proof that the new estimator is unbiased and can have reduced variance, and 4) A demonstration of the approach in the context of Adaptation-Based Programming, showing significant benefits compared to standard policy-gradient without side information.

In what follows, Section 2 describes the standard model for sequential decision processes and reviews a well-known policy gradient learning algorithm, which is the basis of our approach. In Section 3, we formalize the notion of "influence" and then in Section 4 describe a simple modified policy gradient algorithm that takes this information into account. Section 5 then analyzes the bias and variance of the new algorithm. Next we describe the application of the new approach in the context of adaptation-based programming (Section 6) followed by experimental results in this domain (Section 7). Finally, we discuss related work and conclude.

2. MDP's and Policy Gradient Methods

We consider sequential decision making in the framework of Markov Decision Processes (MDPs) (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 2000). An MDP is a tuple $\{X, A, P, R, P_0\}$ where X is a set of possible states, A is the set of actions, and P is a transition probability function giving the probability $\Pr(x_{t+1}|x_t, a_t)$ of a transition to state x_{t+1} when action a_t is taken in state x_t. Finally, R is a reward function that maps states to real-valued rewards and P_0 is a distribution over initial states. In this work, we are interested in solving large MDPs where the states are represented in terms of M state variables. Thus, the state at time t will be denoted by an M-dimensional vector $x_t = [x_t^1, x_t^2, .., x_t^M]$. We will also denote the decision made at time t as d_t, which can take the

value of any action. For the rest of the paper, we use upper case symbols to denote random variables (e.g. X_t or D_t) and lower-case for concrete values (e.g. x_t or d_t).

Each decision D_t is made by a policy π_θ, which is a stochastic function from X to A that is parameterized by a vector $\theta \in \mathbb{R}^K$. Our approach will not make any assumptions about π_θ other than that it is a differentiable function of θ. For example, it might be a log-linear probability model or a neural network over a set of state features. Executing π_θ for a horizon of T steps starting in initial state x_0 drawn from P_0 produces a random trajectory of states, decisions, and rewards $H = \{X_0, R_0, D_0, X_1, R_1, .., D_t, X_T, R_T\}$ and we will denote the expected sum of rewards on such trajectories as $\eta(\theta) = \mathbf{E}\left(\sum_t R_t\right)$. The learning goal we consider in this work is to find policy parameters that maximize the value of $\eta(\theta)$. We do this in an RL setting where the MDP model is unknown and the algorithm must learn via direct interaction with the environment.

Policy Gradient RL. We consider a policy-gradient RL approach (Williams, 1992; Baxter and Bartlett, 2001), where the main idea is to estimate the gradient $\eta(\theta)$ with respect to the parameters θ and to then adjust the parameters in the direction of the gradient. The primary difficulty lies in estimating the gradient via direct interaction with the environment for which there is a significant literature. We now review the well-known likelihood ratio gradient estimator, which is the basis of our approach (Williams, 1992).

Given any trajectory $h = \{x_0, r_0, d_0, x_1, ..., d_T, x_T, r_T\}$ of states x_i, decisions d_i, and rewards r_i we let $r(h) = \sum_{t=0}^{T} r_t$ denote the sum of rewards along the trajectory and $q_\theta(h)$ denote the probability of generating h using policy π_θ. The policy value can now be written as,

$$\eta(\theta) = \sum_h q_\theta(h) r(h)$$

where the summation is over all length T trajectories. Taking the gradient of this expression and applying the likelihood ratio method we get the standard form for $\nabla\eta(\theta)$ (Williams, 1992).

$$\nabla\eta(\theta) = \mathbf{E}\left[\sum_{t=0}^{T-1} \frac{\nabla\pi_\theta(D_t|X_t)}{\pi_\theta(D_t|X_t)} \sum_{j=t+1}^{T} R_t\right] \tag{1}$$

where X_t, D_t, and $R_t = R(X_t)$ are random variables that respectively denote the state, decision, and reward at time t when following π_θ. The utility of this expression is that it allows for a simple unbiased estimator of the gradient to be calculated without knowledge of the model. This is done by estimating the expectation with an average taken over N independent trajectories $\{h(1), h(2)..., h(N)\}$ obtained by following π_θ in the environment as follows,

$$\hat{\nabla}\eta(\theta) = \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{t=0}^{T-1} \frac{\nabla\pi_\theta(d_{n,i}|x_{n,i})}{\pi_\theta(d_{n,i}|x_{n,i})} \sum_{j=t+1}^{T} r_{n,j}\right) \tag{2}$$

where $d_{n,i}$, $x_{n,i}$, and $r_{n,i}$ are respectively the i'th decision, state, and reward of trajectory h_n. For large horizons T this gradient estimate can have large variance, which is commonly addressed by using a discount factor $\beta \in [0, 1)$ in order to reduce the credit given to decisions for distant future rewards, yielding the following modified estimator.

$$\hat{\nabla}\eta(\theta) = \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{t=1}^{T-1} \frac{\nabla\pi_\theta(d_{n,i}|x_{n,i})}{\pi_\theta(d_{n,i}|x_{n,i})} \sum_{j=t+1}^{T} \beta^{j-t-1} r_{n,j}\right) \tag{3}$$

For $\beta = 1$ this estimator is identical to the above and hence unbiased. However, as β decreases, the variance is reduced but bias can grow, which often has an overall positive impact on computing a useful gradient direction.

3. Context-Specific Independence

In this section, we introduce the type of side information that we wish to incorporate into a policy gradient estimator. At a high level, this side information consists of context-specific conditional independencies between the state, decision, and reward variables in the domain. Graphical models such as tree-based dynamic Bayesian networks (DBNs) (Boutilier et al., 2000) or acyclic decision diagrams (ADDs) (Hoey et al., 1999) have been commonly employed to compactly represent MDP models by directly encoding context-specific independencies among variables. The common idea behind these models is that given the context of a state x_{t-1} and decision d_{t-1} at some time $t-1$, for any state variable V_t, the models identify a set of *parent variables* $\text{Pa}(V_t|x_{t-1}, d_{t-1})$, which can be any variables from time-step $t-1$.[1] For reward or decision variable V_t, the context is simply x_t and the parents are denoted by $\text{Pa}(V_t|x_t)$. The intention is that the parents given a context are the only variables that matter with respect to specifying the distribution over values of V_t, with all other variables being conditionally independent. For instance, for our example in Section 1, the parents of the location variable Y_t change depending on the value of O_{t-1}, which indicates whether the time step is odd or even. If $O_{t-1} = 1$ then no action can change the location (it is a "press" time step) and the parents are $\{Y_{t-1}, O_{t-1}\}$ indicating that these are the only variables that the value of Y_t depends on. Otherwise, when $O_{t-1} = 0$ a move action may change the location yielding a parent set $\{Y_{t-1}, O_{t-1}, D_{t-1}\}$, indicating that the decision at $t-1$ can influence the value of Y_t.

Since our approach is not tied to a particular type of model such as DBN or ADD, we formulate the notion of independence in terms of a generic parent function, which must satisfy certain general properties with respect to the underlying MDP.

Definition 1 *For state variable V_t, a parent function $Pa(V_t|x_{t-1}, d_{t-1})$ is* consistent *with an MDP and parameterized policy representation π_θ if the following two conditions hold:*

1. *For any context (x_{t-1}, d_{t-1}), there are no cycles in the graph of variables where an arc is drawn to V_t from every variable in $Pa(V_t|x_{t-1}, d_{t-1})$.*
2. *For any context (x_{t-1}, d_{t-1}), $\Pr(V_t|x_{t-1}, d_{t-1}) = \Pr(V_t|Pa(V_t|x_{t-1}, d_{t-1}))$, where the probability is with respect to the MDP transition function.*

Corresponding conditions exist when V_t is a decision or reward variable where the context is now x_t and the parent function is given by $Pa(V_t|x_t)$. Also, in the second condition, when V_t is a decision variable, the probability is with respect to the policy representation.

Importantly, our approach will not require explicit access to a consistent parent function, but rather only access to a function related to it, which we now formalize. Consider the set of all possible

1. In most prior work, e.g. (Boutilier et al., 2000; Hoey et al., 1999), decision variables are not explicitly included in the graphical model, meaning that the set of their parents is implicitly assumed to be all state variable (i.e. the policy considers all state variables). Rather, we allow decision variables to have context specific parent functions, where the parents can depend on the specific values of state variables. This allows for policy architectures that need not look at all state variables and where this set (the parents) can vary from state to state.

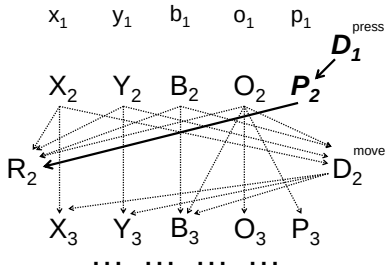


Figure 1: The conditional influence graph for "press" decision variable D_1 is shown after the state at $t = 1$ has been observed. Note that D_1 has only a single path of influence $\{D_1, P_2, R_2\}$, as shown by the thick arcs. Thus "press" decisions can only influence the immediate reward whereas move decisions like D_2 have at least one path to every subsequent reward (not shown) via the state variables X_3, Y_3 and B_3.

state, decision, and reward variables from time $t = 0$ to $t = T$ and assume that we have observed a trajectory up to time t resulting in state x_t. We are interested in defining the set of future variables that can possibly be "influenced" by the decision D_t at time t given x_t. For this purpose, we construct the following *conditional influence graph*. Draw an arc from variable D_t to any state or reward variable V_{t+1} at time $t + 1$ such that $D_t \in \text{Pa}(V_{t+1}|x_t, D_t = a)$ for some value a of D_t. These are all the variables at time $t + 1$ whose value can possibly be influenced by the choice of D_t. Next for all future times $t' > t + 1$ and for any state, decision, or reward variable $V_{t'}$ we draw an arc to $V_{t'}$ from any variable in $\bigcup_{x,a} \text{Pa}(V_{t'}|X_{t'-1} = x, D_{t'-1} = a)$. Thus, there is an arc from a parent to a child for times $t' > t + 1$ if the parent can potentially influence the child, i.e. there is some way of setting the previous state and decision so that the parent is influential. Figure 1 shows a part of the conditional influence graph for the example problem given earlier[2]. Using this graph, we now define the *reward influence function* $I(t, x_t, t')$ for a parent function Pa() as follows:

- $I(t, x_t, t') = 0$, if given x_t there is *no path* in the influence graph for Pa() from D_t to $R_{t'}$
- $I(t, x_t, t') = 1$, otherwise

From this definition, it is clear that if $I(t, x_t, t') = 0$ then the choice of D_t at time t given x_t can have no influence on the value of the reward at time t'. Our algorithm only requires access to such an influence function, and correctness can be guaranteed even when given access to only an approximation. In particular, we say that an approximate reward influence function $\hat{I}(t, x_t, t')$ is *influence complete* for an MDP if there exists a consistent parent function for the MDP such that if $I(t, x_t, t') = 1$ then $\hat{I}(t, x_t, t') = 1$. That is, the approximate influence function will always indicate a potential influence when one exists. Stated differently if an influence function $\hat{I}$ is influence complete then it soundly detects when a reward at time t' is independent of a decision at time t, which is the key property that our algorithm will require. Importantly, while computing the true influence function I may often be difficult or even undecidable, obtaining an influence complete approximation is often straightforward. For example, in our application of adaptation-based programming, an influence function can be computed via program analysis, or if a qualitative model is available that provides an estimate of the parent function, influence can be computed via path finding.

2. State variable P is introduced so that the rewards are state-dependent only whereas the original description in Section1 had state-action dependent rewards. This is done only to maintain consistency with the policy gradient RL formulation in Section 2. P is 1 only in the time-step immediately after a button is pressed and 0 in all other cases.

4. Policy Gradient with Influence Functions

Suppose that we are given an influence function $\hat{I}(t, x_t, t')$ as a form of side information. Intuitively, such a function can be used to help alleviate the credit assignment problem faced by a policy gradient estimator. To see how this might be done, consider Equation 1 which shows that the exact policy gradient can be expressed as an expectation over a sum of terms, one for each decision from $t = 0$ to $t = T - 1$. Each term is simply the likelihood-scaled gradient of the policy's decision at time t weighted by the sum of rewards after that decision is made. Essentially, this weighting strategy assumes that a decision is equally responsible for the value of any future reward. Given this view, if our influence function tells us that the decision at time t is independent of a future reward, then it is plausible that we can soundly drop that reward from the weighting factor of term t. This results in the following modified gradient expression.

$$\nabla \eta_{SI}(\theta) = \mathbf{E}\left[\sum_{t=0}^{T-1} \frac{\nabla \pi_\theta(D_t|X_t)}{\pi_\theta(D_t|X_t)} \sum_{j=t+1}^{T} \hat{I}(t, X_t, j)\ R_j\right] \tag{4}$$

As proven in the next section, it turns out that when the influence function is influence complete this expression is equal to the true gradient. Thus, we can obtain an unbiased estimator by sampling trajectories and averaging, arriving at the estimator.

$$\hat{\nabla} \eta_{SI}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \left(\sum_{t=0}^{T-1} \frac{\nabla \pi_\theta(d_{n,i}|x_{n,i})}{\pi_\theta(d_{n,i}|x_{n,i})} \sum_{j=t+1}^{T} \hat{I}(t, x_{n,t}, j) r_{n,j} \right) \tag{5}$$

The new estimator $\hat{\nabla} \eta_{SI}(\theta)$ simply disregards the contribution to the gradient estimate of decision-reward pairs that are judged as being independent according to $\hat{I}$. When this judgement is correct, the new estimator can significantly simplify credit assignment. When such terms are not ignored, as is the case for standard gradient estimates, it is much more difficult to efficiently and accurately estimate the gradient due to the "noise" introduced by those terms. Overcoming such noise necessitates the use of many more trajectories.

Incorporating Discounting. To help reduce variance we now consider an analogue of the biased gradient estimate from Equation 3 that takes the influence function into account. Equation 4 reduces variance by discounting rewards with respect to a decision based on the duration between the decision and reward. This is based on the assumption that decisions nearer to a reward are likely to be more influential. However, this assumption can be strongly violated when there are many non-influential actions for certain rewards. For instance, in our running example, move actions which always influence future rewards are discounted more due to the intermediate button presses, all but one of which are irrelevant to a given reward. In general, if there are t time-steps between a relevant decision d and a reward r, then d sees a discounted reward of $\beta^t r$ but if n of those decisions are irrelevant, an intuitively better discount factor would be $\beta^{t-n} r$. That is, a more appealing assumption is to discount based on the number of potentially relevant decisions between a decision and a reward, rather than the total number of decisions as in Equation 4.

Using the conditional influence graph described in the previous section, we can define a biased estimator that captures the above intuition. Let $L(t, x_t, j)$ be the number of decisions on the shortest path between D_t and R_j in the influence graph conditioned on x_t and let $\beta(t, j) = \beta^{L(t, x_t, j)}$ be a modified discounting factor. Using this factor in place of the standard discounting approach of Equation 3 results in a biased gradient estimate, a necessary consequence of discounting. The intention is that the reduction in variance will result in a net benefit with respect to estimating a useful gradient direction.

Our complete learning algorithm, which incorporates discounting, denoted as PGRI (policy gradient with reward influences) is given in Algorithm 1. This algorithm is an incremental implementation of the gradient estimator and when $\beta = 1$, it corresponds to the unbiased version of the estimator (Equation 5) since updates are only performed when $\hat{I}(i, x_i, j) = 1$. The algorithm updates parameters at every time-step rather than after N trajectories. This is a common practice in policy gradient RL since the number of parameter updates for the episodic update is quite small compared to the amount of accumulated experience, leading to slow learning, particularly for long trajectories. However, while the sum of the updates made by these online algorithms between recurrent states is in the gradient direction, the individual updates are not in expectation (see discussion in Baxter and Bartlett (2001)). Nevertheless, while the theoretical understanding of such updates are less well understood, in practice they very often yield superior performance.

Algorithm 1 PGRI algorithm.

{**Inputs:** At time step t, observed reward r_t, prior trajectory h, influence-complete $\hat{I}$, current param. vector θ, discount factor β, learning rate α
Output: Updated parameter vector }
$\Delta\theta = 0$
for i = 1 : t **do**
  {d_i is the i'th decision in h}
  if $\hat{I}(i, x_i, j) = 1$ **then**
    $\Delta\theta = \Delta\theta + \frac{\nabla\pi_t(d_i)}{\pi(d_i)} r_t \beta^{L(i,x_i,j)}$
    {$L(i, x_i, j)$ is the number of decisions on the shortest path between d_i and r_j}
  end if
end for
$\theta = \theta + \alpha\Delta\theta$

5. Algorithm Properties

We now consider the properties of the new estimator when $\beta = 1$, which corresponds to $\hat{\nabla}\eta_{SI}(\theta)$. First, we show that this estimator is unbiased, a property which follows from the result given next.

Theorem 2 *For any reward influence function $\hat{I}$ that is influence complete with respect to the underlying MDP, the following holds:* $\nabla\eta_{SI}(\theta) = \nabla\eta(\theta)$.

Proof Let $\mathbf{pa}$ denote a consistent parent function for which $\hat{I}$ is influence complete with respect to the underlying MDP (see Section 3). Let $\eta_t(\theta)$ be the expected value of the single reward R_t at time t when following policy π_θ. Thus $\eta(\theta) = \sum_{t=1}^{T} \eta_t(\theta)$. The sum over length t trajectories h_t can be written as a sequence of sums over the individual state and decision variables,

$$\begin{aligned}
\eta_t(\theta) = \mathbf{E}[R_t] &= \sum_{h_t} q_\theta(h_t) r_t \\
&= \sum_{x_0^1} q(x_0^1) \sum_{x_0^2} q(x_0^2) \ldots \sum_{d_0} \pi_\theta(d_0|x_0) \sum_{x_1^1} q(x_1^1|x_0, d_0) \ldots \sum_{x_t^M} q(x_t^M|x_{t-1}, a_{t-1})\, r_t(x_t) \qquad (6) \\
&= \sum_{x_0^1} q(x_0^1) \sum_{x_0^2} q(x_0^2) \ldots \sum_{d_0} \pi_\theta(d_0|\mathbf{pa}) \sum_{x_1^1} q(x_1^1|\mathbf{pa}) \ldots \sum_{x_t^M} q(x_t^M|\mathbf{pa})\, r_t(\mathbf{pa}) \qquad (7)
\end{aligned}$$

where $q(x_i^m|\mathbf{pa}) = \Pr(x_i^m|\mathbf{pa})$ and for clarity we have not shown the arguments to the parent function. Equation 7 is the result of applying the properties of consistent parent functions to Equation 6. Taking the gradient of Equation 7 w.r.t. θ gives us $(t+1) \cdot M$ sums for $t+1$ states and another t sums for the decision variables. Since the probabilities of state transitions are not parameterized by θ, $\nabla q(x_i^m|\mathbf{pa})$ is zero for all i and m and we are left with a sum of t terms where a single $\pi_\theta(d_i|x_i)$ appears differentiated in each term,

$$\nabla \eta_t(\theta) = \sum_{i=0}^{t-1} \sum_{x_0^1} q(x_0^1) \sum_{x_0^2} q(x_0^2) ... \sum_{d_i} \nabla \pi_\theta(d_i|\mathbf{pa}) \sum_{x_1^1} q(x_1^1|\mathbf{pa}) ... \sum_{x_t^M} q(x_t^M|\mathbf{pa})\ r_t(\mathbf{pa}) \tag{8}$$

The crucial step in this proof is based on the observation that given history upto the decision D_i, we may re-order the sums over the remaining terms as long as we use a topological ordering of the variables w.r.t. the conditional influence graph. Using the arc-drawing procedure from Section 3 that was used to define the influence graph and reward influence function, we can partition the variables after D_i into two sets: U which includes those variables that are unreachable from D_i and $\neg U$ for the rest. Denote by $\nabla \eta_t^i(\theta)$ the i'th component of $\nabla \eta_t(\theta)$ where $\nabla \eta_t(\theta) = \sum_{i=0}^{T-1} \nabla \eta_t^i(\theta)$.

Given the above definitions we can derive that the variables in U denoted by $u_1, u_2, ...$ can be summed over before the variables in $\neg U$ denoted by $\neg u_1, \neg u_2, \ldots$ provided the variables in each set are topologically ordered w.r.t. the conditional influence graph. That is,

$$\nabla \eta_t^i(\theta) = \sum_{h_i} q(h_i) \sum_{u_1} q(u_1|\mathbf{pa}) \sum_{u_2} ... \sum_{d_i} \nabla \pi_\theta(d_i|\mathbf{pa}) \sum_{\neg u_1} q(\neg u_1|\mathbf{pa}) \sum_{\neg u_2} q(\neg u_2|\mathbf{pa}) ...\ r_t(\mathbf{pa})$$

where the first sum is over the history h_i up to time i. This holds based on the following argument. Since the conditional influence graph is acyclic, there must exist a topological ordering of the variables in U which we denote by $\{U_1, U_2, ...\}$. Similarly, we have a topological ordering of the variables in $\neg U$. For any variable $U_i \in U$, the probability distribution $q(U_i|\mathbf{pa})$ becomes well defined given h_i and values for the variables in U that precede U_i in the topological ordering [3] and correspondingly for the variables in $\neg U$. Since the reordering of the summation terms satisfy such an ordering and the parent function is consistent with the MDP, all the probability terms remain well-defined (and unchanged) which is all that is needed for the equation to hold.

Now consider the case where the influence complete influence function asserts $\hat{I}(i, x_i, t) = 0$. This implies r_t is unreachable from D_i in the influence graph given x_i. Also, by the definition of $\neg U$, r_t is unreachable from every variable in $\neg U$. Therefore, when $\hat{I}(i, x_i, t) = 0$, r_t is conditionally independent of D_i and every variable in U given h_i and values of the variables in $\neg U$. This allows us to move r_t to the left of d_i in the summation as shown,

$$\nabla \eta_t^i(\theta) = \sum_{h_i} q(h_i) \sum_{u_1} q(u_1|\mathbf{pa}) \sum_{u_2} ...\ r_t(\mathbf{pa}) \sum_{d_i} \nabla \pi_\theta(d_i|\mathbf{pa}) \sum_{\neg u_1} q(\neg u_1|\mathbf{pa}) \sum_{\neg u_2} ... \tag{9}$$

The summations over the $\neg u_i$ now simply become 1. Further, since π_θ is a probability distribution over decisions we have that,

$$\sum_{d_i} \nabla \pi_\theta(d_i|\mathbf{pa}) = \nabla \sum_{d_i} \pi_\theta(d_i|\mathbf{pa}) = \nabla 1 = 0.$$

This shows that the entire sum must be zero. Thus, $\hat{I}(i, x_i, t) = 0$ implies $\nabla \eta_t^i(\theta) = 0$ and hence,

$$\nabla \eta_t(\theta) = \mathbf{E}\left[\sum_{t=0}^{T-1} \frac{\nabla \pi_\theta(D_t|X_t)}{\pi_\theta(D_t|X_t)}\ \hat{I}(t, X_t, t)\ R_t\right] \tag{10}$$

3. For the decision variables in U, $\pi_\theta(d_i|\mathbf{pa})$ becomes well-defined.

where we have re-written Equation 8 in the likelihood ratio form. Finally, it is a straightforward manipulation to combine the gradient estimates for the reward at each time-step into the full gradient.

$$\nabla\eta(\theta) = \sum_{t=1}^{T} \nabla\eta_t(\theta) = \nabla\eta_{SI}(\theta). \tag{11}$$

■

Corollary 3 *For any reward influence function $\hat{I}$ that is influence complete with respect to the underlying MDP, the estimator in Equation 5 is an unbiased estimate of $\nabla\eta(\theta)$.*

Variance Reduction. We now consider the variance of the new estimator for $\beta = 1$ compared to the standard estimator that ignores the influence function. From Equations 2 and 5 we see that for any set N of independently sampled trajectories H, the gradient estimate $\hat{\nabla}\eta(\theta)$ can be written as the sum of $\hat{\nabla}\eta_{SI}(\theta)$ and a residual term $\text{Res}(\theta)$ which is the sum of all terms in $\hat{\nabla}\eta(\theta)$ that are ignored by our new estimator (i.e. when $\hat{I} = 0$). Note that the dependence on H is left implicit. From this we get the following relationship between the variances of the two estimators.

$$\text{Var}\left(\hat{\nabla}\eta(\theta)\right) = \text{Var}\left(\hat{\nabla}\eta_{SI}(\theta) + \text{Res}(\theta)\right) = \text{Var}\left(\hat{\nabla}\eta_{SI}(\theta)\right) + \text{Var}\left(\text{Res}(\theta)\right) + 2\text{Cov}\left(\hat{\nabla}\eta_{SI}(\theta), \text{Res}(\theta)\right)$$

The variance of $\nabla\eta_{SI}(\theta)$ will be less than that of the standard estimator $\nabla\eta(\theta)$ as long as,

$$\text{Var}\left(\text{Res}(\theta)\right) > -2\text{Cov}\left(\hat{\nabla}\eta_{SI}(\theta), \text{Res}(\theta)\right).$$

Since the variance of the residual is always positive this will only be violated when there is a strong negative correlation between the residual and $\nabla\eta_{SI}(\theta)$. While it is possible to construct pathological MDPs where this can occur, it is very difficult to find a realistic MDP where this occurs. In typical domains that we have considered it is the case that the magnitude of the residual variance dominates the magnitude of the correlation term, which implies reduced variance for the new estimator. For example, in all of our applications domains it is not hard to verify that dominating negative correlation does not occur.

6. Adaptation-Based Programming (ABP)

The potential benefits of incorporating influence information are clearly visible in the application domain of Adaptation-Based Programming (ABP), which is therefore the focus of our experimental results. ABP (sometimes called "partial programming") integrates reinforcement learning (RL) into a programming language with the intention of allowing programmers to represent uncertainty in their programs. ABP permits specific decisions in the *adaptive program* to be left unspecified. Instead, the programmer provides a reward signal as a measure of program performance, which the ABP system uses to learn a policy for the open decision points. Our work is built on an existing ABP Java library (Bauer et al., 2011), where the main construct is the `adaptive` object, which represents an open decision point. Given some information (context) representing the current program state, the adaptive can suggest an action, chosen from a set of specified actions. Another way to view an adaptive is a mapping from a context to an action which needs to be learned over time in order to optimize an objective. Figure 2 contains simple examples of adaptive programs reproduced

```
if (test()) {
  if (A.suggest()) {
    reward(1);
  } else {
    reward(2);
  }
} else {
  if (B.suggest()) {
    reward(2);
  } else {
    reward(1);
  }
}
  Program P1
```

```
a = A.suggest();
b = B.suggest();

if (test()) {
  if (a) {
    reward(1);
  } else {
    reward(2);
  }
} else {
  if (b) {
    reward(2);
  } else {
    reward(1);
  }
}
  Program P2
```

```
for (i=1; i<N; i++) {
  c = randomContext();
  m = move.suggest(c);
  reward(payoff(c,m));
}
     Program P3
```

Figure 2: Illustrative adaptive programs reproduced from (Pinto et al., 2010). `test`, `randomContext`, and `payoff` are non-adaptive methods. `reward` and `suggest` are part of the ABP library (Bauer et al., 2011).

from (Pinto et al., 2010). P1 is a conditional structure where `A,B` are adaptives representing uncertainty in how to pick a branch. `A` and `B` have fixed context and boolean actions. The `suggest` method returns either `true` or `false`. The notion of optimizing a performance objective motivates the second important construct: `reward(r)`. In order to measure program performance, ABP allows the programmer to place reward statements anywhere in the program, where numerically larger rewards indicate better performance. For example, in P1, the reward statements measure the desirability of each path from root to leaf. (Bauer et al., 2011) contains a detailed description of the syntax and semantics of each ABP construct and examples of realistic adaptive programs.

Learning Problem: Given an adaptive program P, we want to learn a *choice function* for each adaptive in P, which is a mapping from an adaptive's context type to its action type. A *policy* π for P is a set of choice functions, one for each adaptive in P. Executing P on input x using policy π results in a sequence of `suggest` calls where each call to adaptive A is handled by its choice function in π. Also encountered are a sequence of deterministic `reward` statements. Let $R(P, \pi, x)$ denote the sum of these rewards. Assuming input x is drawn from some probability distribution D, the goal of learning is to find a policy π that maximizes $\mathbf{E}_D[R(P, \pi, x)]$, the expected sum of rewards[4] over a single execution of P. Computing the optimal value of π analytically is infeasible for non-trivial adaptive programs. Instead, the ABP system attempts to learn a good, possibly optimal, policy through multiple executions of P with inputs drawn from D. We now illustrate the potential for leveraging influence information in ABP and review recent work (Pinto et al., 2010) on the topic.

Motivation and Prior Work: Consider the adaptive programs in Figure 2. P1 is a conditional structure with two adaptives (`A` and `B`) at the inner nodes and rewards at the leaves. This is an easy problem to solve optimally for even simple RL algorithms, since on each execution a single `suggest` call is made to one of the adaptives and a single reward observed. Now consider P2 which is identical except the choices are made at the very top of the program and stored in `a` and `b` which are used later. On every execution, the learner now sees two choices, one of which is spurious.

4. This assumes that each execution terminates in a finite number of steps resulting in a finite reward sum. If this is not the case, we can instead maximize the *discounted* infinite sum of rewards.

This is a harder credit assignment problem since the learner cannot see which branch was actually taken but only the sequence of adaptive calls and rewards. However, P1 and P2, both valid ABP programs, are equivalent from the *programmer's point of view*. A very different learning problem is induced by P3 which is a simple bounded loop. On each iteration, it makes a single choice based on a randomly generated context, producing a reward. Each reward is generated independently and only depends on the most recent decision. However, a standard learning algorithm has no way of detecting this and instead sees a very noisy reward signal containing the sum of future rewards, which dramatically complicates the credit assignment problem.

The solution proposed by Pinto et al. (2010) is based on the observation that there is information useful for simplifying credit assignment in the structure of the adaptive program itself. Their method relies on existing program analysis techniques that extract program structure information, abstracted as an oracle that can compute two properties, *value dependence* and *path irrelevance*, of an adaptive program during its execution. In order to leverage this analysis as *side information* for an RL process, the SARSA(λ) RL algorithm (Sutton and Barto, 2000) was modified so that the eligibility parameters were sensitive to the side information. As one example, the eligibility of a decision was reset to zero whenever it becomes path independent of the current execution point of the program. While this approach was shown to significantly outperform SARSA(λ) without the use of side information, the semantics of the algorithm are unclear. Further, even when restricted to just policy evaluation, the updates to eligibility values fail to satisfy a condition necessary for convergence (Bertsekas and Tsitsiklis, 1996). In general, the algorithm lacks theoretical characterization and is a somewhat ad-hoc solution to the problem of using side information about program structure in RL.

A more fundamental issue concerns the use of value function-based learning algorithms in ABP systems. Previous work such as ALISP (Andre and Russell, 2000, 2002) provides convergence guarantees, but only by making strong assumptions about the abstractions used to represent value functions at choice points. In practice, for programs written by non-RL experts, it will be rare to have such conditions satisfied and there will typically be substantial partial observability for the learner. In such cases, value, or TD-based methods, can be problematic. Thus, in this work we consider policy gradient-based methods, which are arguably a more principled way for dealing with partial observability, which is inevitable in ABP.

Policy Gradient for ABP: Since we assume that our adaptive programs always terminate, it is straightforward to apply the policy gradient algorithms presented in Sections 2 and 4. The process being controlled corresponds to the adaptive program with decision points occurring whenever an adaptive is reached. The underlying process state corresponds to the entire program state (stack, memory, counter, etc) and the observations correspond to the context made available by the programmer at each adaptive. We assume that the context c of each adaptive A is a user defined feature vector that is computed each time the adaptive is encountered. For example, if the decision corresponds to making a particular choice in a game, then the context would encode game features relevant to that choice.

As discussed previously, a policy π for an adaptive program is the set of choice functions and as such we parameterize our policies by a parameter vector θ which is the concatenation of parameter vectors for each choice function. The parameters corresponding to adaptive A are denoted by θ_A, which are further assumed to have components specific to each action a, denoted by $\theta_{A,a}$. We use a standard log-linear stochastic policy $\pi_\theta(a, A, c)$ which defines the probability of selecting action a at adaptive A given context c and parameters θ as,

$$\pi_\theta(a, A, c) = \frac{\exp(c \cdot \theta_{A,a})}{\sum_{a'} \exp(c \cdot \theta_{A,a'})} \tag{12}$$

Given this functional form, we can directly apply the likelihood-ratio approach for estimating the performance gradient $\eta(\theta) = \mathbf{E}_D[R(P, \pi_\theta, x)]$ based on a set of trajectories generated by executing the program using parameters θ. These estimates can then be used to conduct stochastic gradient descent by taking steps in the estimated gradient directions.

We are yet to discuss how we construct an influence complete approximation of the reward influence function for the PGRI algorithm. This is actually the easiest part since it turns out that we can use the same program analysis from (Pinto et al., 2010) to define a suitable approximation. The easy transformation of program analysis information into an (approximate) influence function is not a coincidence as a close correspondence seems to exist between the conditional influence graph over an MDP and the control-flow/ data-flow graph structure computed during program analysis. As an example of the program analysis capabilities, it is able to infer that in program P2 the decision made by adaptive B does not influence the final reward in program trajectories where `test()` evaluates to true as desired. Following the approach in (Pinto et al., 2010), we annotate the adaptive programs by hand with the same outputs that the analysis would produce. Implementing the program analysis engine is a large but straightforward software engineering task which is ongoing.

Combining the program analysis with the PGRI algorithm results in the first ABP system for non-ML-experts that is both provably convergent and leverages program structure for faster learning. Our experiments on large adaptive programs demonstrate the utility of taking influence information into account.

7. Experiments

We compare the performance of the PGRI algorithm against the baseline approach in Equation 3 on a set of adaptive programs. We evaluate β-dependence, convergence rate and the performance of the learned policy. Each adaptive program is executed multiple times during which it learns. After every 10 executions, we switch off learning and evaluate the learned behavior on 10 executions (for programs with stochastic rewards) and report the total reward obtained during evaluation, during which the standard practice of executing the highest probability action is used. Finally, all results are averaged across 10 runs to eliminate random effects and initialization bias. The learning rate for both algorithms is set to 10^{-5} and kept constant throughout. We now briefly describe the adaptive programs and associated credit assignment issues.

Sequence Labeling (SeqLabel): An important problem in ML, sequence labeling involves the labeling of each character of an input string. Here, we use input sequences of length 20, generated by a Hidden Markov Model. Both character and label sets have cardinality 10. The training set contains 500 labeled sequences while the test set contains 200. The adaptive program uses a single adaptive to suggest a label for each input in each sequence in the dataset, using the character and previous label as context. A reward of $+1$ is given after each correct prediction. Each program run consists of one pass through the dataset. Ideally, the programmer would indicate an independency between labels for consecutive sequences via an ABP library method. However, our informal studies have shown that programmers routinely fail to identify such independencies or forget to indicate them resulting in a noisy reward signal. Our algorithm is unaffected by this since program analysis detects that all the choices made during the labeling of one sequence either fall out of scope or get over-written at the end of a sequence, terminating their influence on future decisions.

Multi-label sequence labeling (NetTalk): Here, each input sequence has multiple label sequences. We use a subset of the NetTalk dataset (Sejnowski and Rosenberg, 1987) which consists

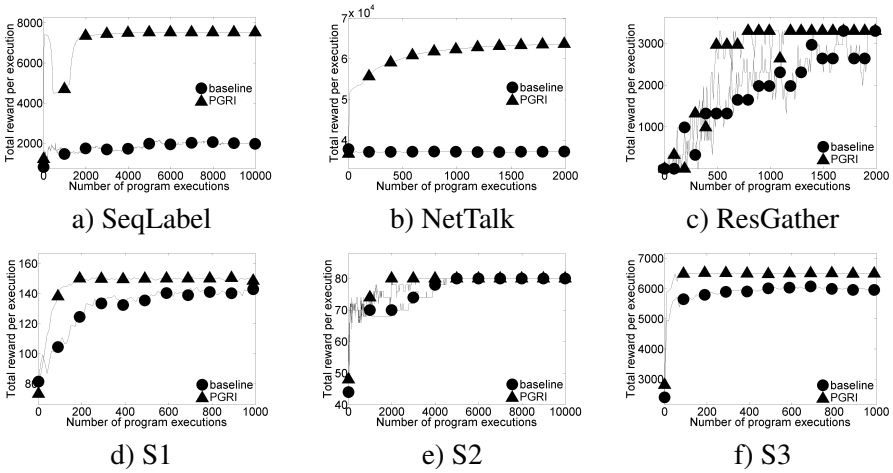
of English words and label sequences for 1) the phonemes used to pronounce each word and, 2) the stresses placed on it. There are 50 phonemes and 5 stresses. Instead of predicting labels directly, the program uses output coding (Hao and Fern, 2007) as follows: Every label symbol is encoded using 10 bits and a different adaptive predicts the bit at each position giving a total of 20 adaptives. The context for the adaptive at bit position i consists of two sliding windows of length five, one over the input sequence and the second over the previous predictions at position i. A reward of $+1$ is given for each correct bit prediction. Both training and test sets contain 500 sequences with some overlap. This program poses a harder credit assignment problem since the two label predictions are now inter-leaved besides the inter-sequence noise.

Resource gathering (ResGather): We consider the resource gathering problem in a Real-Time Strategy (RTS) game where a number of peasants must collect and deposit as much gold as possible within a fixed amount of time. The map consists of a grid with a single base and mine at fixed locations. Each peasant must move to the mine and dig there and return to the base to deposit the gold. A single adaptive controls all the peasants. After all the peasants have acted, the "world" returns a vector of the amounts of gold deposited by each peasant. Although the optimal policy is this scenario is obvious, the learning problem is challenging even for a small number of peasants and map since the rewards are zero almost everywhere and the decision sequences are interleaved which makes credit assignment very hard. Program analysis can associate a peasant with the deposited gold, which our algorithm is able to exploit to perform better credit assignment.

Synthetic Programs: We also experiment with three synthetic adaptive programs adapted from (Pinto et al., 2010). The first program S1 is the loop program P3 where N is set to 1000 and the adaptive can have one of 10 contexts. The second program S2 consists of two decision trees in sequence with choices at the root and internal nodes and stochastic rewards at the leaves. All three decisions are made at the top leading to partial observability. The third synthetic program S3 is more complex and consists of multiple nested trees and loops. The outer structure is a tree containing decision trees in some branches and loops in others. Stochastic rewards of different magnitudes are scattered all over the program. Taken together, these programs represent common usage patterns in adaptive programs.

Effect of discarding irrelevant rewards: Figure 3 shows the performance of PGRI compared to the baseline method (PG) without discounting for all six programs. Our algorithm clearly does better than the baseline on all programs both in terms of convergence rate and the eventual learned performance, indicating the usefulness of side information. On problems where the primary source of noise in the reward signal comes from the independence between reward streams (SeqLabel, NetTalk), the baseline fails completely. While S1 and S3 also present a similar challenge, they are somewhat easier learning problems and the baseline is able to learn, though significantly more slowly than PGRI. The remaining programs (S2, ResGather) do not contain this kind of noise and the baseline eventually learns a policy as good as PGRI.

Effect of discounting: Next we evaluate the effect of varying β for a number of values. The results are shown in Figure 4 for two representative values of β and three programs that show interesting trends to avoid cluttering the figures. The remaining programs are relatively robust to values of β that are not too high. The figures clearly show the baseline's sensitivity to the value of β. This is expected since the different programs pose "contrary" credit assignment problems where a large discount factor helps for some programs and hurts for other programs. For example, in the NetTalk programs, updating choices based on a large number of future rewards makes the problem harder while the opposite is true for the resource gathering problem. The baseline quickly

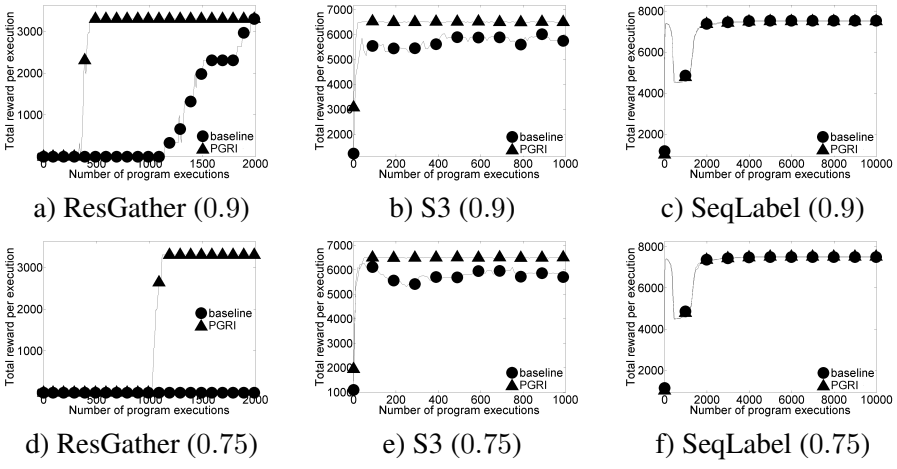


a) SeqLabel b) NetTalk c) ResGather

d) S1 e) S2 f) S3

Figure 3: Performance graphs of adaptive programs with no discounting. The y axis is the total reward over a single execution and the x axis shows the number of program executions.

becomes invariant on the SeqLabel (and NetTalk) programs as β decreases from one. However, it fails completely on ResGather even for the moderately high value of $\beta = 0.75$. Across all programs, the PGRI algorithm is more robust to β than the baseline, a desirable quality since users of ABP systems (non-experts in RL) are not expected to be able to tune the learning algorithm.



a) ResGather (0.9) b) S3 (0.9) c) SeqLabel (0.9)

d) ResGather (0.75) e) S3 (0.75) f) SeqLabel (0.75)

Figure 4: Performance graphs of representative adaptive programs for two moderately high values of β and a subset of adaptive programs. The remaining programs are either invariant to moderately high values of β (S1, S2) or qualitatively similar to the ones shown (NetTalk $\approx$ SeqLabel). For SeqLabel, both methods perform identically and the graphs overlap.

Finally, Figure 5 shows the classification accuracy before and after learning on the NetTalk training set and the final evaluation on the test set since the real objective in this domain is to obtain accurate sequence labels, not binary labels. As expected, the baseline fails on $\beta = 1$ but learns as

(a) Phoneme ($\beta = 1$)

	Before	After	Test-Set
baseline	16.2	15.1	14.0
PGRI	17.4	55.1	54.2

(b) Phoneme ($\beta = 0.75$)

	Before	After	Test-Set
baseline	18.7	45.0	44.6
PGRI	17.1	49.0	47.4

(c) Stress ($\beta = 1$)

	Before	After	Test-Set
baseline	48.2	33.2	32.5
PGRI	42.3	80.4	80.1

(d) Stress ($\beta = 0.75$)

	Before	After	Test-Set
baseline	49.6	81.0	80.9
PGRI	45.36	81.0	80.2

Figure 5: Label prediction accuracy (as percentage) for phoneme and stress labels in the NetTalk datasets. Shown are accuracies before and after learning on the training set and the final evaluation on the test set.

well as our algorithm for $\beta = 0.75$. Our method is practically invariant to β on this problem. While stresses are eventually predicted accurately, predicting phonemes is a much harder problem due to the large number of possible labels and learning is very slow. While better phoneme prediction accuracy is needed, it is important to note that a relatively simple adaptive program is able to do reasonably well on an important and challenging problem.

Overall, these results clearly demonstrate the benefits of our method over a standard policy gradient method. Although we do not show it, the PGRI method demonstrates considerably lower variance across learning runs which is related to smaller variance in its estimates.

8. Related Work

The notion of leveraging structure in domain variables is not new. Much work has been done in the planning community towards planning algorithms for compact representations of large MDP's (Boutilier et al., 2000; Hoey et al., 1999). However, these approaches are generally intractable in practice, since the underlying planning problems are computationally hard. There has been very limited work on incorporating information about variable and reward influences into RL algorithms. Recent work by Tamar et al. (2011) assumes partial knowledge of the transition function (rather than just influence information) and shows how to incorporate that information into policy-gradient RL yielding strong guarantees of reduced variance. The influence information that we use is a very different way of representing partial information about a model. An interesting future direction is to combine these different types of knowledge into one estimator.

The most closely related work is the TD-based iSARSA(λ) algorithm given in (Pinto et al., 2010), where ABP was also the motivating application. That algorithm incorporates influence information into the SARSA(λ) algorithm in a largely ad-hoc way. A direct comparison between our approach and that work is only possible on the smallest of our test problems since their method uses Q-tables without any function approximation. In those cases, our method and theirs perform comparably. More fundamentally, while it is possible that some modified version of their approach would be competitive on larger programs, the SARSA-based approach has no guarantees of convergence, unlike our policy-gradient formulation. Further, we are able to provide such guarantees while making few assumptions about the architecture of the parameterized policy, which in the case of ABP corresponded to arbitrary adaptive programs. In general, there is no clear understanding of

when TD methods such as SARSA might empirically outperform policy gradient methods and it is customary in the literature (Loch and Singh, 1998; Baxter and Bartlett, 2001) to study improvements within each class independently, which we do here for the policy-gradient approach.

9. Summary

This paper introduced the idea of incorporating side information about influences among variables and rewards into policy-gradient estimation. We formalized the notion of influence and provided an algorithm for using the information. The algorithm was proven to be unbiased and conditions were given for when the new estimator will have reduced variance. We demonstrated our new algorithm in the substantial application domain of adaptation-based programming, where influence information can be automatically computed via standard program analysis techniques. The results showed significant speedups in learning compared to standard policy gradient.

References

David Andre and Stuart J. Russell. Programmable reinforcement learning agents. In *NIPS*, pages 1019–1025, 2000.

David Andre and Stuart J. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pages 119–125, 2002.

Tim Bauer, Martin Erwig, Alan Fern, and Jervis Pinto. Adaptation-based programming in java. PEPM, pages 81–90, 2011.

Jonathan Baxter and Peter L. Bartlett. Infinite-horizon policy-gradient estimation. *JAIR*, 15:319–350, 2001.

D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations* 1. *Artificial Intelligence*, 121(1-2):49–107, 2000.

Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *ICML*, pages 118–126, 1998.

Guohua Hao and Alan Fern. Revisiting output coding for sequential supervised learning. In *IJCAI*, 2007.

J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. Spudd: Stochastic planning using decision diagrams. In *UAI*, pages 279–288, 1999.

J. Loch and S. Singh. Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *ICML*, 1998.

Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, pages 278–287, 1999.

Jervis Pinto, Alan Fern, Tim Bauer, and Martin Erwig. Robust learning for adaptive programs by leveraging program structure. In *ICMLA*, pages 943–948, 2010.

Terrence J Sejnowski and Charles R Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1(1):145–168, 1987.

Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2000.

Aviv Tamar, Dotan Di Castro, and Ron Meir. Integrating partial model knowledge in model free rl algorithms. In *ICML*, pages 305–312, 2011.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, May 1992.