Policy Search via the Signed Derivative

J. Zico Kolter and Andrew Y. Ng
Computer Science Department, Stanford University
{kolter,ang}@cs.stanford.edu

***Abstract*— We consider policy search for reinforcement learning: learning policy parameters, for some fixed policy class, that optimize performance of a system. In this paper, we propose a novel policy gradient method based on an approximation we call the *Signed Derivative*; the approximation is based on the intuition that it is often very easy to guess the *direction* in which control inputs affect future state variables, even if we do not have an accurate model of the system. The resulting algorithm is very simple, requires no model of the environment, and we show that it can outperform standard stochastic estimators of the gradient; indeed we show that Signed Derivative algorithm can in fact perform as well as the *true* (model-based) policy gradient, but without knowledge of the model. We evaluate the algorithm's performance on both a simulated task and two real-world tasks — driving an RC car along a specified trajectory, and jumping onto obstacles with an quadruped robot — and in all cases achieve good performance after very little training.**

I. Introduction

In this paper we consider policy search for reinforcement learning. In this setting, one considers a parametrized control policy and then, by interacting with the environment, modifies the parameters to optimize some cost function. For example, if our control task was to drive a car along a desired trajectory, the cost function could penalize deviations from the trajectory, and the control policy could determine the steering and throttle as a simple (say, linear) function of current state features; in this domain, the policy search task would involve learning the coefficients on the state features to obtain a low cost (i.e., follow the trajectory well). We focus in particular on the policy gradient approach, where we optimize the cost function using gradient descent with respect to the policy parameters.

While there exist many different methods for approximating the policy gradients, in this paper we propose a new algorithm that makes use of what we call the *Signed Derivative* approximation. This method allows us to compute an approximation to the policy gradient *without* a model of the system. Our algorithm is based on the following simple insight: the only term in the policy gradient that depends on the dynamics model is the *derivative* of future state elements with respect to the control inputs. However, while these true derivatives are difficult to compute, we claim that often it is very easy to estimate the *sign* of many of these derivatives; that is, we only want to know the general *direction* of how control adjustments will affect the state. Consider again the example of the car mentioned earlier, where for instance one of the state variables is lateral deviation from the desired trajectory, and one of the controls is the steering angle. While it may be very difficult to know the true derivative of future lateral deviations with respect to the steering angle, the sign of the derivative is in fact quite obvious: turning more to the left typically results in a lateral deviation that is also more to the left. While such "obvious" derivative signs clearly don't apply to all control tasks, we demonstrate in this paper that they do apply in many interesting domains; in such situations, we show that we can drastically improve the performance of policy gradient methods by using these signed derivatives. Indeed, we show that in many cases, the Signed Derivative method not only outperforms standard stochastic policy gradient algorithms (such as the REINFORCE [15] family of algorithm), but actually performs as well as the true (model-based) policy gradient algorithm, but without any knowledge of the model, only the ability to simulate a single trace.

The remainder of this paper is organized as follows. In Section II we present preliminary material and describe the general Signed Derivative algorithm more formally. In Section III we present theoretical results. In Section IV we present empirical results for the algorithm on a number of different domains, both simulated and real-world. Finally, in Section V we discuss related work, and conclude the paper in Section VI.

II. The Signed Derivative Algorithm

A. Preliminaries and Notation

We consider control in a Markov Decision Process (MDP), which is a tuple $M = (S, A, T, H, C)$, where S is a set of states, A is a set of actions, T is the (unknown, but temporarily assumed to be deterministic) system dynamics $T : S \times A \rightarrow S$, H is a time horizon and C is a known (one-step) cost function $C : S \times A \rightarrow \mathbb{R}$. Since we are concerned with general, continuous state and action domains, we let $S \subseteq \mathbb{R}^n$ and $A \subseteq \mathbb{R}^m$. We can capture time-varying dynamics and costs by including time as a state variable, though for the remainder of this paper we will make any time dependence explicit. Finally, although our algorithm is extendable to general cost functions, for the sake of concreteness we will here assume a common quadratic form of the reward function

$$C_t(s_t, u_t) = (s_t - s_t^\star)^T Q_t (s_t - s_t^\star) + u_t^T R_t u_t$$

where $s_t^\star$ denotes the desired state of system at time t, and Q_t and R_t are diagonal positive semidefinite matrices that penalize state deviation and control respectively.

A (time-dependent) policy $\pi : S \times \mathbb{R} \rightarrow A$ is a mapping from states and times to actions. As we are focused on the policy-search setting in this paper, here we consider policies parametrized by some set of parameters θ — we use the notation $u = \pi(s, t; \theta)$ to denote the policy π, parametrized by θ, evaluated at state s and time t. For example, a common class of policies that we will consider in this paper is policies that are linear in the state features

$$u = \pi(s; \theta) = \theta^T \phi(s, t)$$

where $\phi : S \times \mathbb{R} \to \mathbb{R}^k$ is a mapping from states and times to features and $\theta \in \mathbb{R}^{m \times k}$ is a set of parameters that linearly map these features into controls.

Given a policy, we define the *multi-step cost* function (also called the value function, or just the cost function, as opposed to the one-step cost function defined above above) as the sum of all one-step costs over the horizon H,

$$J(s_0, \theta) = \sum_{t=1}^{H} C_t(s_t, u_{t-1})$$

where $u_t = \theta^T \phi(s_t, t)$ and where $s_{t+1} = T(s_t, u_t)$. We can now more formally define the policy gradient algorithm as a gradient descent method that repeatedly updates the parameters according to

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(s_0, \theta)$$

where α is a step size and $\nabla_\theta J(s_0, \theta)$ is the gradient of the cost function with respect to the policy parameters. Although, computing this gradient term can be quite complicated without a model of the system, in the next section we describe a simple approximation method.

B. The Signed Derivative Approximation

In this section we derive a simple approximation to the policy gradient, using an approximation we called the *signed derivative*. We want to emphasize that the final form of the algorithm, shown in Algorithm 1, is quite simple, even though the derivation is somewhat involved.

To motivate the signed derivative method, we first consider the basic question of *why* ones needs a model of the system to compute the policy gradient $\nabla_\theta J(s, \theta)$. We will derive this result shortly, but it turns out that the policy gradient depends on the model only through terms of the form

$$\left(\frac{\partial s_t}{\partial u_{t'}}\right)$$

for $t > t'$. These terms are the *Jacobians* of future states with respect to previous inputs. They provide the critical motivation for the signed derivative approximation, so it is worth looking at them more closely. These Jacobians are matrices $\left(\frac{\partial s_t}{\partial u_{t'}}\right) \in \mathbb{R}^{n \times m}$ where the i, j element of the $\left(\frac{\partial s_t}{\partial u_{t'}}\right)$ denotes the derivative of the ith element of the state s_t with respect to the jth element of $u_{t'}$, i.e.,

$$\left(\frac{\partial s_t}{\partial u_{t'}}\right)_{ij} \equiv \frac{\partial (s_t)_i}{\partial (u_{t'})_j}.$$

In other words $(\frac{\partial s_t}{\partial u_{t'}})_{ij}$ indicates how the ith element of s_t would change if we made a small adjustment to the j element of the control at a previous time t' (and assuming we are following the policy θ).

In general, the elements of these Jacobians are quite difficult to compute, as they depend on the true dynamics model of the environment and the policy parameters θ. However, the signed derivative approximation is based on the insight that often times it is fairly easy to guess the *signs* of the dominant entries of these matrices: this only requires knowing the general *direction* of how previous control inputs will affect future states. Returning to the example of driving a car, it may be very difficult to determine the derivative of a future state with respect to the steering wheel, but the *direction* of the gradient seems fairly obvious: turning the wheel more to the left will likely result in future states also more to the left.

Furthermore, there is another property of these Jacobians that allows us to come up with a reasonable approximation: in many control settings, each state is primarily affected by only *one* control input. For example, if we are driving our car along a straight line, one state variable (for example, the distance traveled along the line), would be primarily affected by only one control (in this case, the gas pedal). Indeed, many control tasks seem to be expressly designed such that this is the case. For example, imagine trying to drive a car where both the steering wheel and gas pedal controlled some different combinations of both the wheel angle and the throttle; while such a control system is technically "equivalent" to a standard car, it would take much more work to learn. This suggests, at least anecdotally, that humans also exploit these orthogonal control effects, and so we can expect many control tasks to be designed in this way. In other words, we can expect one element in each row of the Jacobians to be larger than the others, corresponding to the "dominant" control element, and these are precisely those elements where we can guess their sign.

Given this discussion, the signed derivative approximation is quite straightforward. We approximate *all* the Jacobian terms with a single matrix $S \in \mathbb{R}^{n \times m}$, called the signed derivative, where entries in S correspond to the signs of the dominant entries in the Jacobians (which, by our discussion above, means that S has only one non-zero entry per row). Consider one last time the driving example, and suppose that the car is facing primarily along the x axis. If we represent the state of the car as its position and orientation (x, y, θ), and let u_1 and u_2 be the throttle and steering angle respectively. Then a reasonable estimate for the signed derivative would be

$$S = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}.$$

For instance, $S_{11} = 1$ means that the first state variable (x) is primarily controlled by the first control input (throttle). This makes sense, since the car is mostly aligned with the x axis, so throttle will primarily affect this state. Similarly, y and θ are primarily affected by the second control (steering), which also makes sense, because the steering wheel can cause the car to both turn and veer to the side.

Finally recall, from the beginning of this section, that the Jacobians were the only terms in the policy gradient that

Algorithm 1 Policy Gradient w/ Signed Derivative (PGSD)

```
Input:
  $S \in \mathbb{R}^{m \times n}$: signed derivative matrix
  $H \in \mathbb{Z}_+$: horizon
  $Q_t \in \mathbb{R}^{n \times n}, R_t \in \mathbb{R}^{m \times m}$: diagonal cost function matrices
  $\alpha \in \mathbb{R}_+$: learning rate
  $\phi : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^k$: feature vector function
  $\theta_0 \in \mathbb{R}^{k \times m}$: initial policy parameters
Repeat:
  1. Execute policy for $H$ steps to obtain
     $u_0, s_1, \ldots, u_{H-1}, u_H$.
  2. Compute approximate gradients w.r.t. controls:
     $\widetilde{\nabla}_{u_t} J(s_0, \Theta) \leftarrow \sum_{t'=t+1}^{H} S^T Q_{t'}(s_{t'} - s^\star_{t'}) + R_t u_t$
  3. Update parameters:
     $\theta \leftarrow \theta - \frac{\alpha}{H} \sum_{t=0}^{H-1} \phi(s_t, t)(\widetilde{\nabla}_{u_t} J(s_0, \Theta))^T$
```

required a model. Therefore, after making such the signed derivative approximation we can now perform (approximate) policy gradient *without* the need for any model of the system. This is precisely the method that we show in Algorithm 1. The precise form of the gradient updates is derived in the next section, but the basic idea of the algorithm is simple: we are just performing policy gradient, replacing the Jacobian terms with the signed derivative approximation S. It may seem surprising that the method would perform well, given that the signed derivative is a very crude approximation to the true Jacobians; but, we will show, from both a theoretical and empirical perspective, that we can expect the algorithm to perform well in many situations.

C. Formal Derivation of the Policy Gradient

Here we prove the claim made in the previous section, that the policy gradient depends only on the dynamics model through the Jacobian terms, and we derive the precise form of the gradient given in Algorithm 1. The derivation is slightly technical, but the algorithm itself can be understood just from the discussion above.

To avoid certain dependencies, we have to initially consider the gradient of the cost function with respect to H different sets of policy parameters for each time, $\Theta = \{\theta_0, \ldots, \theta_{H-1}\}$. We will then take a gradient step in terms of these parameters, projected back into the space where they are all equal. The gradients are given by

$$\begin{aligned}\nabla_{\theta_t} J(s, \Theta) &= \left(\frac{\partial u_t}{\partial \theta_t}\right)^T \nabla_{u_t} J(s, \Theta) \\ &= \phi(s_t, t)\left(\nabla_{u_t} J(s, \Theta)\right)^T.\end{aligned}$$

using the fact that $u_t = \theta_t^T \phi(s_t, t)$ and that s_t doesn't depend on θ_t. Furthermore, using the definition of J,

$$\begin{aligned}\nabla_{u_t} J(s, \Theta) &= \nabla_{u_t} \sum_{t'=1}^{H} (s_{t'} - s^\star_{t'})^T Q_{t'}(s_{t'} - s^\star_{t'}) + u_t^T R_t u_t \\ &= \sum_{t'=t+1}^{H} \left(\frac{\partial s_{t'}}{\partial u_t}\right)^T Q_{t'}(s_{t'} - s^\star_{t'}) + R_t u_t\end{aligned}$$

This gives a gradient with respect to each θ_i (where, as stated, the only model-dependent terms are the Jacobians $\left(\frac{\partial s_{t'}}{\partial u_t}\right)^T$). Therefore, the gradient of the cost with respect to a *single* θ is equivalent to taking a step in the direction of all these θ_i's then projecting onto the space where $\theta_0 = \theta_1 = \ldots = \theta_{H-1}$. This is accomplished by updating each θ_t according to

$$\theta_t \leftarrow \theta_t - \frac{\alpha}{H} \sum_{t'=0}^{H-1} \nabla_{\theta_{t'}} J(s, \Theta),$$

i.e., the policy gradient with respect to a single parameter θ is also given by

$$\nabla_\theta J(s, \theta) = \frac{1}{H} \sum_{t=0}^{H-1} \nabla_{\theta_t} J(s, \Theta).$$

As stated, the only terms that depend on the model in this sum are the Jacobians. This derivation should make it apparent that Algorithm 1 is simply approximating the policy gradient by substituting the signed derivative for all the Jacobian terms.

III. Theoretical Results

Given that the signed derivative is admittedly a rather crude approximation to the true Jacobians, there remains some question as to why we might expect such an approach to work. While the ultimate test of the algorithm's usefulness is, of course, its empirical performance, the results we present here can give insight and intuition into why we obtain the positive results shown in latter sections. We first describe the basic intuition behind the analysis.

There are many possible sources of error for any policy gradient algorithm, but here we analyze the two types of error introduced by the signed gradient approximation itself. First, the signed gradient allows only one control variable to influence a given state variable; even if a state element is *primarily* affected by one control, there most likely exist smaller influences from the other controls as well. Second, the signed gradient makes no attempt to capture the relative magnitudes (or the magnitudes of any kind) of the entries in the Jacobian. Formally, these approximations are represented as

$$\frac{\partial s_t}{\partial u_{t'}} = D(S + E_{t,t'}) \qquad (1)$$

for the signed derivative S and matrices $E_{t,t'} \in \mathbb{R}^{m \times n}$ and positive diagonal $D \in \mathbb{R}^{n \times n}$. The D matrix scales the entries in the signed derivative, accounting for the second type of error mentioned above. As we will see more formally below, this type of error isn't overly costly, since it has the effect of simple scaling the entries of the cost functions. Especially in the extreme case where policy gradient finds a solution that obtains near-zero cost, the actual entries of Q_t become unimportant.

The $E_{t,t'}$ terms capture other errors: they add arbitrary constants to the entries in the signed derivative, accounting for the effects of additional control inputs on the states and for time-dependent variation in the relative scaling of the Jacobian. In the worst case, there is little that can be done about such errors: if the entries of $E_{t,t'}$ are large, then the gradient approximation using the signed derivative can be very far from the true gradient. However, there is a great deal of reason to believe that, in many situations $E_{t,t'}$ won't be too large: time-varying scaling should be relatively small over short horizons, and from the discussion in the previous section, we expect cross-terms in the Jacobian to be relatively small in magnitude. And as formalized below, if the $E_{t,t'}$ are small, then we expect the signed derivative to perform well.

Theorem 1: Using the notation from (1), suppose $\|E_{t,t'}\|_2 \leq \epsilon$ for all t, t'.[1] Define the modified cost function $\tilde{Q}_t = DQ_t$. Then, given additional technical assumptions (described fully in the appendix), PGSD will converge with probability one to some solution $\tilde{\theta}$ that is "close" to a local minimum of the cost function $J_{\tilde{Q}}(\theta)$, the cost function that uses $\tilde{Q}_t$ (but the same R_t) as the cost matrices:

$$\|\nabla_\theta J_{\tilde{Q}}(\tilde{\theta})\| \leq O(\epsilon).$$

Furthermore, if performing gradient descent with respect to the true gradient (of the actual cost function) results in η-optimal policy parameters — i.e., $J_Q(\theta^\star) \leq \eta$ — then PGSD also obtains an order η-optimal solution[2]

$$J_Q(\tilde{\theta}) \leq \kappa(D)\eta + O(\epsilon).$$

Proof: (sketch) The full proof is given in the appendix[3], but we provide a very brief sketch. The proof proceeds in three steps. First, we show that the gradient approximated using the signed derivative is equivalent to the true gradient using the $\tilde{Q}$ costs, plus a bounded error term

$$\tilde{\nabla}_\theta J_Q(\theta) = \nabla_\theta J_{\tilde{Q}}(\theta) + \tilde{E}$$

for $\|\tilde{E}\| \leq O(\epsilon)$. Second, we show that following this approximate gradient using a stochastic gradient method will converge, with probability one, to a point that is close to a minimum of $J_{\tilde{Q}}(\theta)$. Finally, we show that given suitable assumptions about the region of convergence, a policy that is close to locally optimal for $J_{\tilde{Q}}(\theta)$ will also be close to locally optimal for $J_Q(\theta)$. ■

IV. Experimental Results

A. Simulated Two-Link Arm

While we will present experiments on real systems shortly, we begin by presenting an evaluation of our proposed method on a simulated two-link arm, in order to rigorously compare to previous policy gradient approaches, and to provide a readily available implementation of our approach. Code for the all the results in this section is available at `http://cs.stanford.edu/~kolter/rss09sd`. We emphasize that the purpose of this section is to specifically compare PGSD with other policy gradient approaches. The control task itself is fairly straightforward, and many other approaches such as adaptive control or iterative learning control could also be applied, though this is beyond the scope of this paper; we will discuss these related works more in Section V.

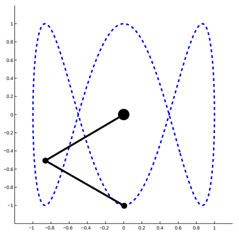
Fig. 1. Two-link pendulum trajectory following task.

The two-link pendulum is a well-known control task in robotics and control. The system, shown in Figure 1 consists of two planar links; the state consists of the joint angles and velocities of both joints and the control specifies a torque at each of the joints. The equations of motion can be easily derived from Lagrangian dynamics, and we introduce stochasticity to the system by adding Gaussian noise to the torques before integrating the equations of motion. The task we consider here, also shown in the figure, is to move the end effector along some desired trajectory. When the model of the system is known, it is fairly easy to apply classical control methodologies such as inverse dynamics or LQR to find an optimal controller, but of course we don't provide this model to PGSD or other comparable algorithms. We feel that this is a particularly demonstrative example for the Signed Derivative algorithm, since it is well-known that there *are* cross terms that cause all joints to be affected by all the control inputs — for instance, a common (more challenging) task is to swing the pendulum upright and balance by applying torques only to the elbow — yet we claim that the Signed Derivative approximation is still reasonable, since joints are *primarily* affected by their own control.

The cost function for this domain penalizes deviations from the desired joint angles (we first computed the trajectory in joint space), and we use a time horizon of $H = 5$. Note that this doesn't mean that the controller only needs to follow the trajectory for 5 steps, but rather that at each time the controller should ideally act optimally with respect to a receding horizon of $H = 5$; since the cost function itself "guides" the arm along the trajectory, such a horizon is suitable. We use a linear control policy $u_t = \theta^T \phi(s_t, t)$ where ϕ contains 1) deviations from desired joint angles, 2) deviations from desired joint velocities, 3) desired joint accelerations, and 4) $\sin(2\pi t/t_{\text{total}})$

[1]The $\|\cdot\|_2$ norm of a matrix is equivalent to its maximum singular value.
[2]$\kappa(\cdot)$ denotes the condition number of a matrix.
[3]Additional material, including the full appendix with this proof, is available at `http://cs.stanford.edu/~kolter/rss09sd`.

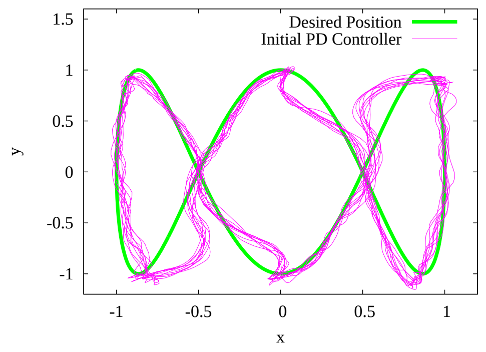

Fig. 2. (top) Trajectory from initial PD controller. (bottom) Trajectories from controller learned using PGSD.

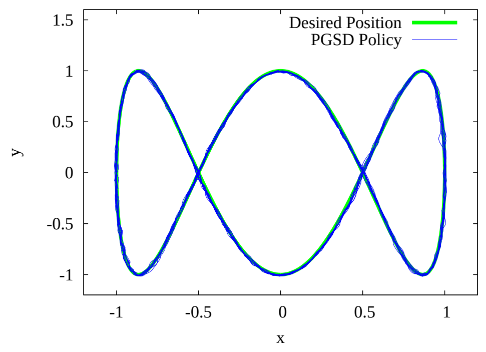

Fig. 3. (top) Trajectory from initial PD controller. (bottom) Trajectories from controller learned using PGSD.

where t_{total} is total time for the complete trajectory (this last term was added to account for a visible periodic pattern in the controls). This leads to s total of 14 parameters for the policy. For algorithms that require a stochastic policy, we added Gaussian noise to the parameters: $u_t = (\theta + \epsilon_t)^T \phi(s_t, t)$, $(\epsilon_t)_{ij} \sim \mathcal{N}(0, \sigma)$.

Figure 4 compares the performance versus time of PGSD, and a well-known policy gradient RL algorithm, the REINFORCE algorithm.[4] All free parameters of the learning algorithms (gradient step sizes, policy noise, number of episodes) were hand-optimized to give that fastest convergence that didn't cause any divergence issues. As the figure shows, PGSD drastically outperform the other methods, converging much faster to a low-cost policy. This improvement is especially notable given that the REINFORCE algorithm is actually given an advantage: since the task we're considering is not episodic (at least not at the time-scale of the horizon), episodic algorithms don't immediately apply, and so we instead allow the algorithm the ability to reset to previous states observed along the trajectory. The REINFORCE without resets in the fiture does not have such an advantage, but also performs much worse. Figures 3 and 2 show the resulting controller learned

[4]We intentionally scaled the parameters of this control task to be the same order of magnitude, so more advanced techniques such as natural gradients[6, 11] didn't improve performance significantly. In preliminary experiments we also evaluated a variety of finite difference and weight perturbation methods, but didn't notice a substantial improvement over REINFORCE for this task.

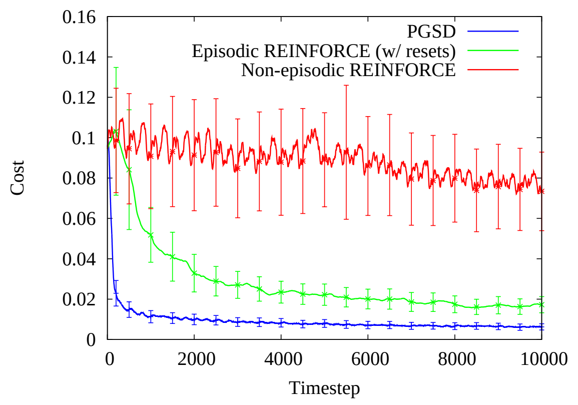

Fig. 4. Average cost versus time for different policy gradient methods. Costs are averaged over 20 runs, and shown with 95% confidence intervals. (Best viewed in color).

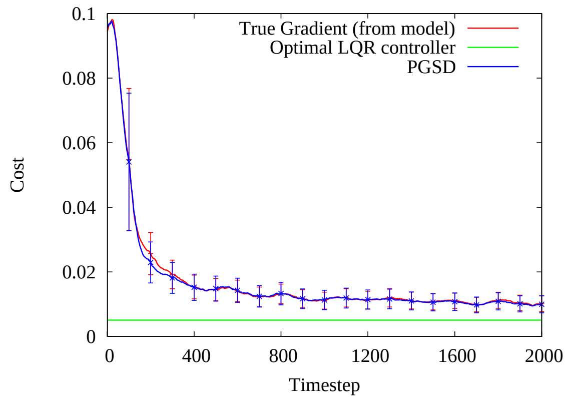

Fig. 5. Average cost versus time for PGSD versus model-based methods. Costs are averaged over 20 runs, and shown with 95% confidence intervals.

by the PGSD algorithm after 2000 time steps (4 times through the trajectory), along with the trajectory achieved by the initial PD controller (used to initialize all the learning algorithms).

We also compare, in Figure 5, the performance of the PGSD algorithm, policy gradient using the true gradient from the model, and an optimal LQR controller. Not surprisingly, the LQR controller performs best: this controller is built by linearizing around the (known) dynamics at each operating point, then computing a series of non-stationary policies for each point (in total, the LQR controller has 9000 parameters). However, using only 14 parameters, the true policy gradient and PGSD algorithm are able to obtain a controller that performs relatively close to this full LQR controller. Furthermore, the most important result is that the learning curve for PGSD is virtually *indistinguishable* from the true policy gradient learning curve; despite the rather crude approximation made by the signed derivative, this resulting algorithm performs *just as well* on this task, and requires no model of the system (and therefore also less computation time, since there is no need for time-consuming finite difference computations).

B. Autonomous RC Driving

In this section we apply the PGSD algorithm to the task of learning to drive an autonomous RC car along a desired

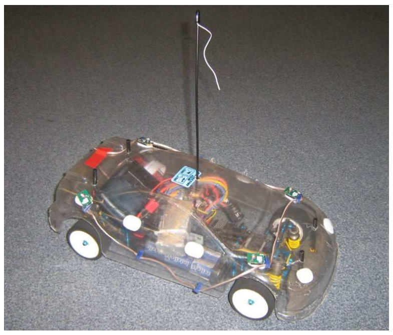

Fig. 6. RC car used for the driving experiments.

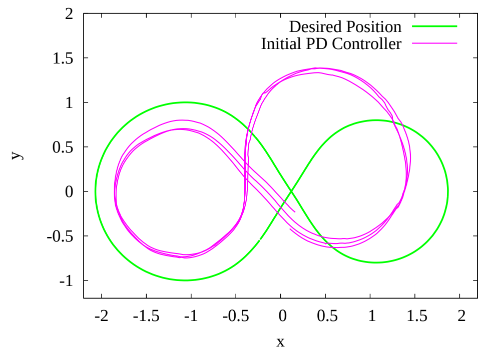

Fig. 7. Desired trajectory for the autonomous RC driving experiments, with trajectory for initial PD controller

trajectory. Figure 6 shows the car, a Tamiya TRF415, which is about 40cm long and 20cm wide. A pattern of LED lights is attached to the car, and tracked by an external PhaseSpace motion capture system for pose estimation. All processing is done on a workstation PC, with controls transmitted to the car at 50hz.

The simplest representation of the car's state is as six dimensional vector representing the 2D position x, y, the orientation θ, and the time derivatives $\dot{x}, \dot{y}, \dot{\theta}$. However, a more natural representation for the signed derivative approach is to represent the car's state relative to some desired trajectory — here the trajectory is specified as a continuous spline that gives the desired state as a function of time. In this alternate representation, the state consists of the longitudinal, lateral, and angular deviation (and their derivatives) from the desired trajectory. The control is two dimensional, consisting of a commanded throttle and steering angle.

We use the same form of linear controller as in the previous sections, but where $\phi(s, t)$ now contains 1) the full state (represented as the deviation terms), 3) the desired velocities, relative to the car frame, 3) the deviations for a target state 0.5 seconds and 4) a constant term. Some of the θ parameters are forced to be zero (so that, for instance, the throttle doesn't depend on the lateral deviation), for a total of 16 parameters in the policy. The cost function penalizes the longitudinal, lateral, and angular deviation, any control outside a specified valid range, and control that changes more that some amount between two time steps (to minimize oscillations). We used a time horizon of $H = 25$.

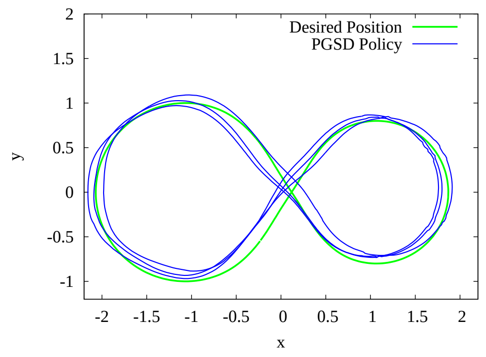

Fig. 8. Desired trajectory for the autonomous RC driving experiments, with typical trajectory learned using PGSD after approximately 20 seconds of learning.

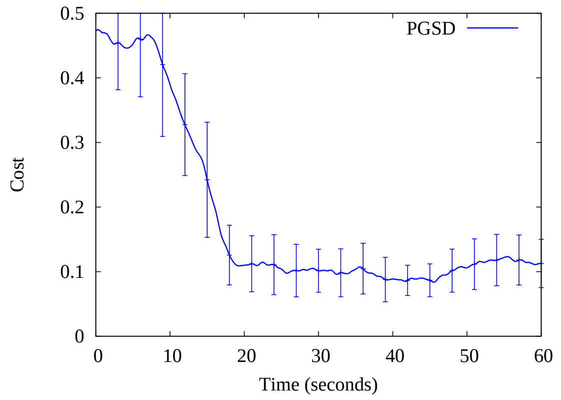

Fig. 9. Average cost versus time for the PGSD algorithm on the RC car task. Costs are averaged over 10 runs, and shown with 95% confidence intervals.

Figures 8 and 7 show the control task we consider: driving the car in an irregular figure-eight pattern at varying speeds (2.0 m/s along the larger loop, 1.5 m/s along the smaller loop). The figure also shows the trajectory followed by an initial PD controller: while the PD controller follows the overall pattern of the trajectory, it clearly does not perform very well. Figure 9 shows the learning curve of the PGSD algorithm. As the figure shows, PGSD is able to very quickly — within an average of 20 seconds, about 3 times around the trajectory — obtain a policy that performs far better than the initial PD controller. We show a typical trajectory from one of these learned controllers in Figure 8. The learned policies do not perform flawlessly — the car still sometimes veers off the desired path — but we feel this is largely due to the limited policy class itself; to perform better, one might need more complex, time-varying policies, to capture the fact that the car needs to behave differently at different points along the path. Nonetheless, PGSD converges to a very reasonable policy — in fact, better than any we were able to hand tune in the same policy class — in just 20 seconds of learning.

C. *LittleDog Jumping*

In this section we present results on applying PGSD to the task of "jumping" the front legs of a quadruped robot to quickly climb up large steps. The "LittleDog" robot that we

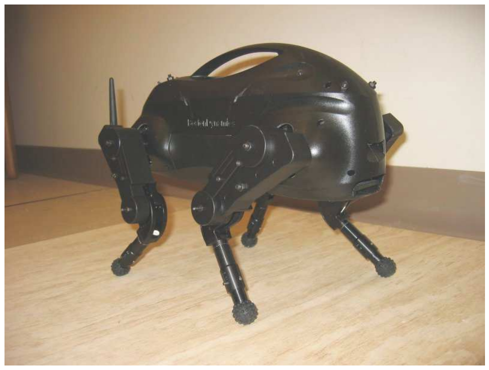

Fig. 10. The LittleDog robot.

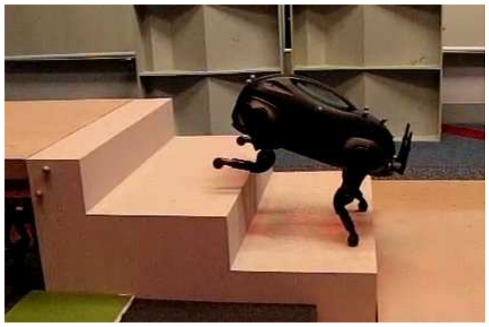

Fig. 11. The desired task for the LittleDog: climb over three large steps.

use for this task, shown in Figure 10, is designed and built by Boston Dynamics, Inc. The task we are concerned with here is shown in Figure 11: we want to quickly climb up three steps, whose height is approximately equal to the robot's ground clearance. Because the steps are so large, the most efficient motion to climb up is to jump the two front legs on a step, then pull the rest of the body up.

However, jumping the front legs on the LittleDog robot is not a trivial task. The LittleDog's legs are not powerful enough to force its body off the ground, so the only means of jumping is to lean the body backwards until the virtually all the mass rests on the hind legs, then quickly raise the front legs and push forward before the robot falls over. Figure 12 shows a properly executed front leg jump. However, if the weight is not shifted properly, the robot will either plant it's feet into the step, or flip over backward. It's very difficult to correct such failures, because usually by the time it is apparent that the robot has failed to jump properly, the robot does not have the power to correct itself. Therefore, jumping is a "one-off" maneuver: we guess an amount to shift backward, then apply an open-loop sequence of joint commands, hoping to jump successfully. The situation is made complicated because the "correct"amount to shift the weight depends, for example, on the state of the robot, namely the current position of the COG relative to the back feet, and the forward velocity of the robot. Because we want a policy that can jump regardless of the initial conditions, we applied the PGSD algorithm to learn a jumping policy that predicts the correct amount to shift given features of the current state.

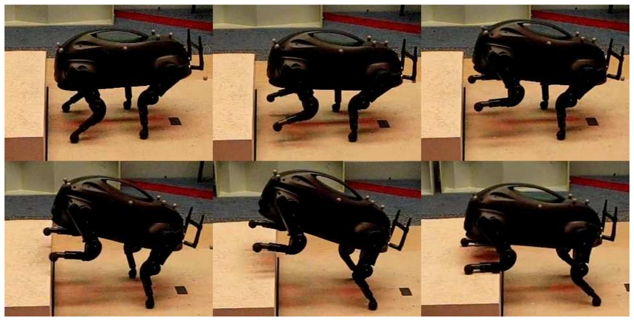

Fig. 12. A properly executed jump.

Although full state space for the LittleDog is 36 dimensional (12 joints and twelve joint velocities plus a 6D pose and 6D pose velocities), we don't need to take into account the complete state. Rather, the only state element that is particularly crucial for the jumping maneuver is the pitch of the body: if the pitch is too small, the dog won't clear the step, but if it is too large, the dog will flip backward. Therefore, the cost function can depend only on the pitch of the dog. The control is a number that indicates how far back to shift the weight before pushing forward; we determine the control as a linear function of three features: 1) the current shift of the center of mass, 2) the forward velocity of the dog and 3) a constant term.

There is one straightforward generalization of the PGSD algorithm, as presented so far, that we make for this task. Although the cost function depends only on the pitch of the robot, it is difficult to know the "optimal" pitch — unlike previous tasks where the optimal state value was clearly defined. Instead, the readily observable quantity is simply whether the jump succeeded, or whether the robot either didn't clear the step or flipped over. Therefore, if we define the one-step cost as the ℓ_1 error between the pitch and optimal pitch, then the gradient is just the sign of the direction we should move our control in. When $H = 1$, the PGSD update then takes on a very simple form:

$$\theta \leftarrow \begin{cases} \theta - \alpha\phi(s) & \text{robot didn't clear step} \\ \theta & \text{jump succeeded} \\ \theta + \alpha\phi(s) & \text{robot flipped backwards} \end{cases}$$

Despite the simplicity of this update rule, it works well in practice. We evaluated this PGSD variant on the LittleDog robot, attempting to climb the three steps as shown in Figure 11. After 28 failures (either flipping backwards or failing to clear the step), the robot successfully jumped all three steps for the first time. After 59 failures, the learning process had converged on a stable controller: the robot succeeded in climbing all three steps for 13 out of the next 20 trials. This is far better than any policy we had been able to code by hand.[5] A video of the learning process on the dog is available at the website mentioned previously.

[5]While it is possible to increase the reliability of the system by adding extra steps to ensure that the robot always enters a similar configuration before each jump, in these experiments we wanted to test precisely how well a controller could perform under many different circumstances.

V. Related Work

As mentioned in the introduction, there is a great deal of work on policy gradient methods for reinforcement learning. If a model of the system is known, then we can compute the gradient using simple finite difference methods — this holds even in stochastic domains if we are allowed to fix the random seeds which lead to this stochasticity, an approach known as the PEGASUS algorithm [10]. These model-based methods have been applied to many robotics domains. However, such a model might not always be available, or might be difficult to learn from data. Additionally, as we have shown, our PGSD method can sometimes perform as well as the model-based methods without any model other than the signed derivative approximation.

In situations where we have no model, we can still apply finite difference methods or weight perturbation, so long as the step sizes are large enough to overcome noise. Such an approach was successfully applied to the task of learning a quadruped trotting gait in [8]. Recently, [13] investigated the effect of sampling distributions on the signal-to-noise ratio of these and similar gradient updates.

A related but different approach uses a likelihood ratio trick to obtain an estimate of the gradient using a number of episodes run under the system and policy of interest: the REINFORCE [15] algorithm was the first of such methods, but many extensions and generalization have been proposed [4, 5, 11, 7]. There has also been work on estimating and using the natural gradient, a gradient that is invariant to reparameterizations of the policy [6, 3, 12]. However, most of these algorithms require running multiple episodes in order to obtain a reasonable estimate of the gradient (or natural gradient), which is difficult for non-episodic tasks such as those we consider. In these domains, PGSD has the strong advantage of only requiring a single episode to obtain an estimate of the gradient.

Our work also shares a strong connection to [1]. This paper proposes a method for using inaccurate simulation models by using only the local gradient information implied by these models. This is quite similar in spirit to our approach, except we discard any need for even an inaccurate simulator, and encode all necessary information directly in the signed gradient: the approximation may be rougher, but unlike this past approach PGSD does not require performing any local policy search in a simulator.

Finally, we want to note the connection between the algorithm we propose here and the field of adaptive control [14, 2] — in particular the subtopics of Model Reference Adaptive Control (MRAC) and Self-Tuning Regulators — and Iterative Learning Control (ILC) [9]. The general philosophy of these approaches is similar to PGSD: they use an error signal (i.e., between the actual and desired state) to directly adapt the parameters. However, typical formulations of MRAC or ILC use hand-crafted update rules to modify the controller, with the focus on analyzing stability properties of the resulting controllers. In contrast, PGSD uses a general update rule that derives completely from the Reinforcement Learning setting of long time horizons and general cost functions, plus the Signed Derivative approximation of the model derivatives. derivatives. Generally speaking however, PGSD could be viewed somewhat as an instance of MRAC or ILC, with a very particular form for the update rule.

VI. Conclusion

In this paper, we proposed the Signed Derivative method, a method for approximating policy gradients, using the insight that often times it is very easy to guess the direction in which control inputs will affect future states. We show that this algorithm, Policy Gradient with the Signed Derivative (PGSD) can perform very well compared to stochastic gradient estimators, and in fact can perform *as well* as the true gradient, even though it has no knowledge of the true environment's model. We further evaluated our algorithm on two real-world control tasks — driving an RC car and jumping with a quadruped robot — and demonstrated very good performance on both domains. While we stress that the PGSD approach is not suitable for all situations (for instance, if the effects of controls on the system is entirely unknown), we feel that in many situations the approach applies quite easily, and offers very substantial performance benefits.

References

[1] Pieter Abbeel, Morgan Quigley, and Andrew Y. Ng. Using innaccurate models in reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, 2006.

[2] Karl Johan Astrom and Bjorn Wittenmark. *Adaptive Control*. Prentice Hall, 1994.

[3] J. Andrew Bagnell and Jeff Schneider. Covariant policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2003.

[4] Jonathan Baxter and Peter L. Bartlett. Infinite-horizon gradient-based policy search. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

[5] Evan Greensmith, Peter L. Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5:1471–1530, 2004.

[6] Sham Kakade. A natural policy gradient. In *Neural Information Processing Systems 14*, 2001.

[7] Jens Kober and Jan Peters. Policy search for motor primitives in robotics. In *Neural Information Processing Systems 21*, 2009.

[8] Nate Kohl and Peter Stone. Machine learning for fast quadrupedal locomotion. In *Proceedings of the AAAI*, pages 611–616, July 2004.

[9] Kevin L. Moore. Iterative learning control: an expository overview. *Applied and Computational Controls, Signal Processing, and Circuits*, 1(1):151–214, 1999.

[10] Andrew Y. Ng and Michael Jordan. Pegasus: A policy search method for large mdps and pomdps. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2000.

[11] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *Proceedings of the IEEE Conference on Intelligent Robotics Systems*, 2006.

[12] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural actor-critic. In *Proceedings of the European Conference on Machine Learning*, 2005.

[13] John W. Roberts and Russ Tedrake. Signal-to-noise ratio analysis of policy gradient algorithms. In *Neural Information Processing Systems 21*, 2009.

[14] Shankar Sastry and Marc Bodson. *Adaptive Control: Stability, Convergence, and Robustness*. Prentice-Hall, 1994.

[15] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.