Using Inaccurate Models in Reinforcement Learning

Pieter Abbeel PABBEEL@CS.STANFORD.EDU
Morgan Quigley MQUIGLEY@CS.STANFORD.EDU
Andrew Y. Ng ANG@CS.STANFORD.EDU
Computer Science Department, Stanford University, Stanford, CA 94305, USA

Abstract

In the model-based policy search approach to reinforcement learning (RL), policies are found using a model (or "simulator") of the Markov decision process. However, for high-dimensional continuous-state tasks, it can be extremely difficult to build an accurate model, and thus often the algorithm returns a policy that works in simulation but not in real-life. The other extreme, model-free RL, tends to require infeasibly large numbers of real-life trials. In this paper, we present a hybrid algorithm that requires only an approximate model, and only a small number of real-life trials. The key idea is to successively "ground" the policy evaluations using real-life trials, but to rely on the approximate model to suggest local changes. Our theoretical results show that this algorithm achieves near-optimal performance in the real system, even when the model is only approximate. Empirical results also demonstrate that—when given only a crude model and a small number of real-life trials—our algorithm can obtain near-optimal performance in the real system.

1. Introduction

In model-based reinforcement learning (or optimal control), one first builds a model (or simulator) for the real system, and finds the control policy that is optimal in the model. Then this policy is deployed in the real system. Research in reinforcement learning and optimal control has generated efficient algorithms to find (near-)optimal policies for a large variety of models and reward functions. (See, e.g., Anderson and Moore (1989); Bertsekas (2001); Bertsekas and Tsitsiklis (1996); Sutton and Barto (1998).)

However, for many important control problems, particularly high-dimensional continuous-state tasks, it is extremely difficult to build an accurate model of the Markov decision process. When we learn a policy using an inaccurate model, we are often left with a policy that works well in simulation (i.e., the policy works well in the model), but not in real-life. In contrast to model-based policy search, there is the other extreme of searching for controllers only on the real system, without ever explicitly building a model. Although successfully applied to a few applications (e.g., Kohl and Stone, 2004), these model-free RL approaches tend to require huge, and often infeasibly large, numbers of real-life trials.

The large number of real-life trials required by model-free RL is in sharp contrast to, e.g., humans learning to perform a task. Consider, for example, a young adult learning to drive a car through a 90-degree turn, more specifically, learning the amount of steering required. On the first trial, she might take the turn wide. She will then adjust and take the next turn much less wide (or maybe even take it short). Typically, it will require only a few trials to learn to take the turn correctly.

The human driver clearly does not have a perfect model of the car. Neither does she need a large number of real-life trials. Instead, we believe that she combines a crude model of the car together with a small number of real-life trials to quickly learn to perform well.

In this paper, we formalize this idea and develop an algorithm that exploits a crude model to quickly learn to perform well on real systems. Our theoretical results show that—assuming the model derivatives are good approximations of the true derivatives, and assuming deterministic dynamics—our algorithm will return a policy that is (locally) near-optimal.

The key idea is to use a real-life trial to *evaluate* a policy, but then use the simulator (or model) to estimate the *derivative* of the evaluation with respect to the policy parameters (and suggest local improvements). For example, if in a car our current policy drives a maneuver too far to the left, then driving in real-life will be what tells us that we are driving too far to the left. However, even a very poor model of the car can then be used to tell us that the change

Appearing in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006.

we should make is to turn the steering wheel clockwise (rather than anti-clockwise) to correct for this error. In particular, we do not need additional real-life trials of turning the steering wheel both clockwise and anti-clockwise in order to decide which direction to turn it. Therefore, even a crude model of the car allows us to significantly reduce the number of real-life trials needed compared to model-free algorithms that operate directly on the real system.

Compared to standard model-based algorithms, our approach has the advantage that it does not require a "correct" model of the Markov decision process. Despite the progress in learning algorithms for building better dynamical models, it remains an extremely difficult problem to model the detailed dynamics of many systems, such as helicopters, cars and some aircraft. The approach we present requires only a very crude model of the system.

Although an extension to stochastic systems might be possible, the algorithm and guarantees we present in this paper only apply to systems that are (close to) deterministic. Despite this limitation, we believe our algorithm still has wide applicability. In our experience with a variety of systems, such as autonomous cars and helicopters, we have observed that they behave close to deterministically (in the absence of external disturbances, such as wind).[1]

Throughout the paper, we assume a continuous state space and a continuous action space.

The remainder of this paper is organized as follows: Section 2 covers preliminaries. Section 3 describes our algorithm in full detail. Section 4 gives formal performance guarantees for our algorithm. Section 5 demonstrates the effectiveness of our algorithm when applied to flying a fixed-wing aircraft in a flight simulator and when applied to driving a real RC car.

2. Preliminaries

A non-stationary Markov decision process (MDP) can be described by a tuple $(S, \mathcal{A}, T, H, s_0, R)$, where $S = \mathbb{R}^n$ is a set of states; $\mathcal{A} = \mathbb{R}^p$ is a set of actions/inputs; $T = \{P_t(\cdot|s,a)\}_{t,s,a}$ is a set of time-dependent state transition probabilities (here, $P_t(\cdot|s,a)$ is the state transition distribution upon taking action a in state s at time t); the horizon H is the number of time steps; s_0 is the initial state at time 0; and $R : S \mapsto \mathbb{R}$ is the reward function. We assume that R is bounded from above, i.e., for all states $s \in S$ we have $R(s) \leq R_{\max}$.

A policy $\pi = (\pi^{(0)}, \pi^{(1)}, \ldots, \pi^{(H)})$ is a set of mappings from the set of states S to the set of actions $\mathcal{A}$ for each time $t = 0, 1, \ldots, H$. The utility of a policy π in an MDP M is given by $U_M(\pi) = \mathrm{E}[\sum_{t=0}^{H} R(s_t)|\pi, M]$. Here the expectation is over all possible state trajectories $s_0, s_1, \ldots, s_H$ in the MDP M. Throughout the paper we consider policies parameterized by a parameter θ. We let $\pi_\theta^{(t)}(s)$ denote the action taken by the policy π_θ when in state s at time t.

This paper focuses on MDPs that are (close to) deterministic. We let $\{f_t : S \times \mathcal{A} \to S\}_{t=0}^{H}$ denote the set of state transition functions. Here, $f_t(s,a)$ gives the expected state at time $t+1$ upon taking action a in state s at time t. For deterministic systems, the system dynamics (T) are specified completely by the time-dependent state transition functions $\{f_t\}_{t=0}^{H}$.

We use $\|\cdot\|_2$ to denote the matrix 2-norm.[2]

3. Algorithm

Our algorithm takes as input an approximate MDP $\hat{M} = (S, \mathcal{A}, \hat{T}, H, s_0, R)$ and a local policy improvement algorithm A. The MDP's dynamics model $\hat{T}$ is a (possibly inaccurate) model of the true dynamics T of the true MDP $M = (S, \mathcal{A}, T, H, s_0, R)$. The local policy improvement algorithm A could be any method that iteratively makes local improvements upon the current policy. Two examples are policy gradient methods and differential dynamic programming.[3]

The algorithm proceeds as follows:

1. Set $i = 0$. Set the initial model estimate $\hat{T}^{(0)} = \hat{T}$.
2. Find the (locally) optimal policy $\pi_{\theta^{(0)}}$ for the MDP $\hat{M}^{(0)} = (S, \mathcal{A}, \hat{T}^{(0)}, H, s_0, R)$ by running the local policy improvement algorithm A.
3. Execute the current policy $\pi_{\theta^{(i)}}$ in the *real* MDP M and record the resulting state-action trajectory $s_0^{(i)}, a_0^{(i)}, s_1^{(i)}, a_1^{(i)}, \ldots, s_H^{(i)}, a_H^{(i)}$.
4. Construct the new model $\hat{T}^{(i+1)}$ by adding a (time-dependent) bias term to the original model $\hat{T}$. More specifically, set $\hat{f}_t^{(i+1)}(s,a) = \hat{f}_t(s,a) + s_{t+1}^{(i)} - \hat{f}_t(s_t^{(i)}, a_t^{(i)})$ for all times t.
5. Use the local policy improvement algorithm A in

[1] Even though stochastic models are often used in reinforcement learning, the stochasticity of the models is often artificial and is to a large part used to capture *unmodeled dynamics*, rather than actual stochasticity.

[2] In the case the matrix is a vector, the matrix 2-norm is equal to the Euclidean norm. In general, the matrix 2-norm is equal to the maximum singular value of the matrix.

[3] Differential dynamic programming iteratively computes a time-varying linear model by linearizing the state transition functions around the current trajectory, and then uses the well-known exact solution to the resulting LQR (linear quadratic regulator) control problem as the step direction to locally improve the current policy. (Jacobson & Mayne, 1970) In its full generality, differential dynamic programming also approximates the reward function by a concave quadratic function. The reward function approximation is irrelevant for this paper: in our experiments the reward functions themselves are already quadratic.

the MDP $\hat{M}^{(i+1)} = (S, \mathcal{A}, \hat{T}^{(i+1)}, H, s_0, R)$ to find a local policy improvement direction $d^{(i)}$ such that $\hat{U}(\pi_{\theta^{(i)}+\alpha d^{(i)}}) \geq \hat{U}(\pi_{\theta^{(i)}})$ for some step-size $\alpha > 0$. Here $\hat{U}$ is the utility as evaluated in $\hat{M}^{(i+1)}$.[4]

6. Find the next (and improved) policy $\pi_{\theta^{(i+1)}}$, where $\theta^{(i+1)} = \theta^{(i)} + \alpha d^{(i)}$, by a line search over $\alpha > 0$. During the line search, evaluate the policies $\pi_{\theta^{(i)}+\alpha d^{(i)}}$ in the real MDP M.
7. If the line-search did not find an improved policy, then return the current policy $\pi_{\theta^{(i)}}$ and exit. Otherwise, set $i = i + 1$ and go back to step 3.

The algorithm is initialized with the policy $\pi_{\theta^{(0)}}$, which is locally optimal for the initial (approximate) model of the dynamics. In subsequent iterations, the performance in the real system improves, and thus the resulting policy performs at least as well as the model-based policy $\pi_{\theta^{(0)}}$, and possibly much better.

In each iteration i, we add a time-dependent bias to the model: the term $s_{t+1}^{(i)} - \hat{f}_t(s_t^{(i)}, a_t^{(i)})$ in step 4 of the algorithm. For the resulting model $\hat{T}^{(i+1)}$ we have that $\hat{f}_t^{(i+1)}(s_t^{(i)}, a_t^{(i)}) = s_{t+1}^{(i)}$ for all times t. Hence, the resulting model $\hat{T}^{(i+1)}$ exactly predicts the real-life state sequence $s_0^{(i)}, s_1^{(i)}, \ldots, s_H^{(i)}$ obtained when executing the policy $\pi_{\theta^{(i)}}$. Thus, when computing the policy gradient (or, more generally, a local policy improvement direction) in step 5, our algorithm evaluates the derivatives along the correct state-action trajectory. In contrast, the pure model-based approach evaluates the derivatives along the state-action trajectory predicted by the original model $\hat{T}$, which—in the case of an imperfect model—would not correspond to the true state-action trajectory.

In contrast to our algorithm, one might also try a model-free algorithm in which real-life trajectories are used to estimate the policy gradient. However, doing so requires at least n real-life trajectories if there are n parameters in the policy.[5] In contrast, our approach requires only a single real-life trajectory to obtain an estimate of the gradient. So long as the estimated gradient is within 90° of the true gradient, taking a small step in the direction of the estimated gradient will improve the policy. Consequently, our approach requires significantly fewer real-life trials than model-free algorithms to learn a good policy.

[4]When the local policy improvement algorithm is policy gradient, we have $d^{(i)} = \nabla_\theta \hat{U}(\theta^{(i)})$, the policy gradient in the MDP $\hat{M}^{(i+1)}$ evaluated at the current policy $\pi_{\theta^{(i)}}$.

[5]If our policy π_θ is parameterized by $\theta \in \mathbb{R}^n$, we can estimate the derivative of the utility with respect to θ_i by evaluating the policy's performance $U(\pi_\theta)$ using parameters θ and again obtain its utility $U(\pi_{\theta+\epsilon e_i})$ using $\theta + \epsilon \cdot e_i$ (where e_i is the i-th basis vector). Then, we estimate the derivative as $\partial U(\pi_\theta)/\partial\theta_i \approx (U(\pi_{\theta+\epsilon e_i}) - U(\pi_\theta))/\epsilon$. In practice, this approach of using finite differences to numerically estimate derivatives not only suffers from the problem of requiring a large number of real-life trajectories, but is also very sensitive to even small amounts of noise in the estimates of the policy's utility.

4. Theoretical Results

Throughout this section *we assume that the real system is deterministic*, and we *assume that the local policy improvement algorithm is policy gradient.* In Section 4.1 we give an informal argument as to why our algorithm can be expected to outperform model-based algorithms that use the same model. Then in Section 4.2 we formalize this intuition and provide formal performance and convergence guarantees.

4.1. Gradient Approximation Error

The dependence of the state s_t on the policy π_θ plays a crucial role in our results. To express this dependence compactly, we define the function $h_t(s_0, \theta)$:

$$\begin{aligned} h_1(s_0, \theta) &= f_0(s_0, \pi_\theta(s_0)), \\ h_t(s_0, \theta) &= f_{t-1}\left(h_{t-1}(s_0, \theta)\right). \end{aligned} \tag{1}$$

Thus $h_t(s_0, \theta)$ is equal to the state at time t when using policy π_θ and starting in state s_0 at time 0. For an approximate model $\hat{T}$ we similarly define $\hat{h}_t$ in terms of the approximate transition functions $\hat{f}_t$ (instead of the true transition functions f_t).

Let $s_0, s_1, \ldots, s_H$ be the (real-life) state sequence obtained when executing the current policy π_θ. Then the true policy gradient is given by:

$$\nabla_\theta U(\theta) = \textstyle\sum_{t=0}^H \nabla_s R(s_t) \left.\frac{dh_t}{d\theta}\right|_{s_0, s_1, \ldots, s_{t-1}}. \tag{2}$$

The derivatives appearing in $\frac{dh_t}{d\theta}$ can be computed by applying the chain rule to the definition of h_t (Eqn. 1).[6] Expanding the chain rule results in a sum and product of derivatives of the transition functions $\{f_t\}_t$ and derivatives of the policy π_θ, which are evaluated at the states of the current trajectory $s_0, s_1, \ldots$.

Let $s_0, \hat{s}_1, \ldots, \hat{s}_H$ be the state sequence according to the model $\hat{T}$ when executing the policy π_θ. Then, according to the model $\hat{T}$, the policy gradient is given by:

$$\textstyle\sum_{t=0}^H \nabla_s R(\hat{s}_t) \left.\frac{d\hat{h}_t}{d\theta}\right|_{\hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{t-1}}. \tag{3}$$

Two sources of error make the policy gradient estimate (Eqn. 3) differ from the true policy gradient (Eqn. 2):

1. The *derivatives* appearing in $\frac{d\hat{h}_t}{d\theta}$ according to

[6]We use the phrasing "derivatives appearing in $\frac{dh_t}{d\theta}$" since for an n-dimensional state-space and a q-dimensional policy parameterization vector θ, the expression $\frac{dh_t}{d\theta}$ denotes the $n \times q$ Jacobian of derivatives of each state-coordinate with respect to each entry of θ.

the model *are only an approximation of the true derivatives* appearing in $\frac{dh_t}{d\theta}$.

2. The *derivatives* appearing in $\nabla_s R$ and in $\frac{d\hat{h}_t}{d\theta}$ *are evaluated along the wrong trajectory*, namely the estimated (rather than the true) trajectory.

In our algorithm, by adding the time-dependent bias to the model (in step 4 of the algorithm), the resulting model perfectly predicts the true state sequence resulting from executing the current policy. This results in the following gradient estimate:

$$\nabla_\theta \hat{U}(\theta) = \sum_{t=0}^{H} \nabla_s R(s_t) \left. \frac{d\hat{h}_t}{d\theta} \right|_{s_0, s_1, \ldots, s_{t-1}}, \quad (4)$$

which evaluates the derivatives along the *true* trajectory. *Thus, by successively grounding its policy evaluations in real-life trials, our algorithm's policy gradient estimates are actually evaluated along true trajectories, eliminating the second source of error.*

We discussed the specific case of policy gradient. However, for any local policy improvement algorithm used in conjunction with our algorithm, we get the benefit of evaluating local changes along the *true* trajectory. For example, our experiments successfully demonstrate the benefits of our algorithm when used in conjunction with differential dynamic programming.

4.2. Formal Results

The following theorem shows that (under certain assumptions) our algorithm converges to a region of local optimality (as defined more precisely in the theorem).

Theorem 1. *Let the local policy improvement algorithm be policy gradient. Let $\epsilon > 0$ be such that for all $t = 0, 1, \ldots, H$ we have $\|\frac{df_t}{ds} - \frac{d\hat{f}_t}{ds}\|_2 \leq \epsilon$ and $\|\frac{df_t}{da} - \frac{d\hat{f}_t}{da}\|_2 \leq \epsilon$. Let the line search in our algorithm be η-optimal (i.e., it finds a point that is within η of the optimal utility along the line). Let all first and second order derivatives of $\{f_t(\cdot,\cdot)\}_{t=0}^H, \{\hat{f}_t(\cdot,\cdot)\}_{t=0}^H, \pi_\theta(\cdot), R(\cdot)$ be bounded by a constant C. Let $R \leq R_{\max}$. Let n, p, q denote the dimensionality of the state space, action space and policy parameterization space respectively. Then our algorithm converges to a locally optimal region, in the following sense: there exist constants K and M—which are expressible as a function only of n, p, q, the constant C, and the MDP's horizon H—such that our algorithm converges to a region where the following holds:*

$$\|\nabla_\theta U(\theta)\|_2^2 \leq 2K^2\epsilon^2 + 2M\eta. \quad (5)$$

For the case of exact line search ($\eta = 0$) we obtain:

$$\|\nabla_\theta U(\theta)\|_2^2 \leq 2K^2\epsilon^2. \quad (6)$$

Proof (sketch). [7] The assumptions on the transition functions $\{f_t\}_{t=0}^H$, the approximate transition functions $\{\hat{f}_t\}_{t=0}^H$, the policy class π_θ and the reward function R imply that there exist constants K and M (expressible as a function of n, p, q, C and H only) such that:

$$\begin{aligned} \|\nabla_\theta U(\theta) - \nabla_\theta \hat{U}(\theta)\|_2 &\leq K\epsilon, & (7)\\ \|\nabla_\theta^2 U(\theta)\|_2 &\leq M. & (8) \end{aligned}$$

Thus, our algorithm performs an iterative gradient ascent (with approximate gradient directions and approximate line search) to optimize the utility function U, which is bounded from above and which has bounded second order derivatives. Our proof proceeds by showing that this optimization procedure converges to a region where Eqn. (5) holds. □

Theorem 1 shows that—if certain boundedness and smoothness conditions hold for the true MDP and the model—our algorithm will converge to a region of small gradient. It also shows that, as the model's derivatives approach the true system's derivatives, the gradients in the (shrinking) convergence region approach zero (assuming exact line search).[8]

In contrast, no non-trivial bound holds for the model-based approach without additional assumptions.[9]

Theorem 1 assumes a deterministic MDP. Space constraints preclude an in-depth discussion, but we note that the ideas can be extended to systems with (very) limited stochasticity.[10]

[7] Due to space constraints we refer the reader to the long version of the paper (Abbeel et al., 2006) for the complete proof.

[8] We note that, like exact gradient ascent, our algorithm could (pathologically) end up in a local minimum, rather than a local maximum.

[9] Consider the MDP with horizon H=1, state-space $S = \mathbb{R}$, transition function $f(s,a) = a$, and $R(s) = C\exp(-s^2)$. Now consider the model with transition function $\hat{f}(s,a) = a + b$, with $b \in \mathbb{R}$. Then, for C and b arbitrarily large, the model-based optimal policy will be arbitrarily worse than the true optimal policy. However, the assumptions of Theorem 1 are satisfied with $\epsilon = 0$, and our algorithm will find the optimal policy in a single iteration.

[10] The results can be extended to those stochastic systems for which we can bound the change in the utility and the change in the values of the derivatives (used in our algorithm) resulting from being evaluated along state trajectories from different random trials under the same policy. We note that the assumption concerns the trajectories obtained under a fixed policy in the policy class π_θ, rather than any policy or sequence of inputs. Feedback can often significantly reduce the diversity of state trajectories obtained, and thus help in satisfying the assumption. Still, the stochasticity needs to be very limited for the resulting bounds to be useful. A significant change in the algorithm (or at least in the analysis of the algorithm) seems necessary to obtain a useful bound for systems with significant

Table 1. Utilities achieved in the flight simulator: mean and one standard error for the mean (10 runs).

Model-based	Our algorithm	Improvement
$-13,028$ (± 330)	$-3,000$ (± 100)	76% ($\pm$2%)

5. Experimental Results

In all of our experiments, the local policy improvement algorithm is differential dynamic programming (DDP). We also used DDP to find the initial policy $\pi_{\theta^{(0)}}$.[11]

5.1. Flight Simulation

We first tested our algorithm using a fixed-wing aircraft flight simulator. The flight simulator model contains 43 parameters corresponding to mass, inertia, drag coefficients, lift coefficients etc. We randomly generated "approximate models" by multiplying each parameter with a random number between 0.8 and 1.2 (one independent random number per parameter).[12] The model with the original (true) parameters is the "real system" in our algorithm. Our (closed-loop) controllers control all four standard fixed-wing aircraft inputs: throttle, ailerons, elevators and rudder. Our reward function quadratically penalizes for deviation from the desired trajectory (which in our case was a figure-8 at fixed altitude), for non-zero inputs, and for changes in inputs at consecutive time steps.[13]

Table 1 compares the performance of our algorithm and the model-based algorithm. Our algorithm significantly improves the utility (sum of accumulated rewards), namely by about 76%, within 5 iterations. Since typically only 1 iteration was required in the line searches, this typically corresponded to (only) 5 real-life trials. Figure 1 shows the result of one representative run: the desired trajectory (black, dotted), the trajectory with the controller from the (approximate) model (blue, solid), and the trajectory obtained after 5 iterations of our algorithm (green, dashed). The top plot shows the (x, y) coordinates (North, East), the bottom plot shows z (the altitude) over time. The initial controller (based on the perturbed model) performs reasonably well at following the (x, y) coordinates of the trajectory. However, it performs poorly at keeping altitude throughout the maneuver. Our algorithm significantly improves the performance, especially the altitude tracking. We note that it is fairly common for badly-tuned controllers to fail to accurately track altitude throughout transient parts of maneuvers. (Such as rolling left-right to stay in the figure-8 for our control task.)

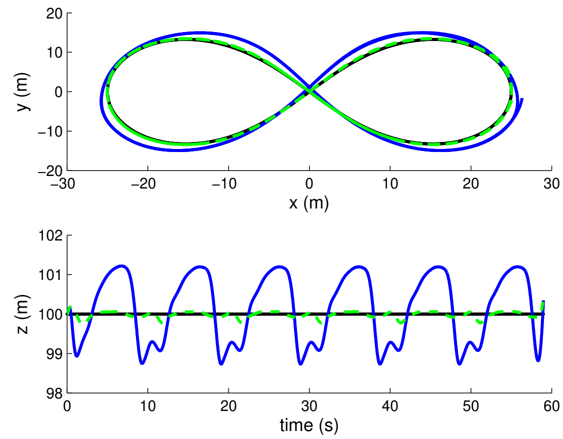

Figure 1. Results of our algorithm for the flight simulator. Black, dotted: desired trajectory; blue, solid: controller from model; green, dashed: controller after five iterations of our algorithm. (See text for details.)

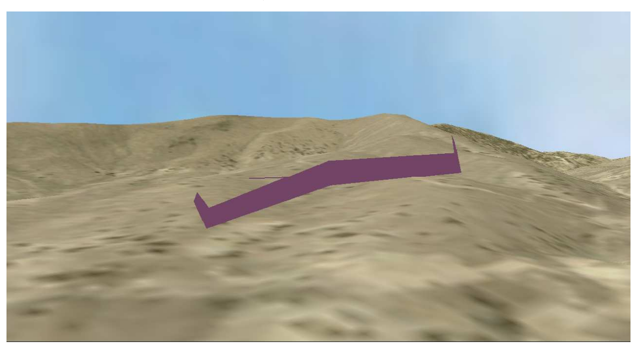

Figure 2. Screenshot of the flight-simulator.

A graphical simulation[14] of the trajectories flown by the model-based controller and by the controller found by our algorithm is available at the following url:

`www.cs.stanford.edu/~pabbeel/rl-videos`.

stochasticity.

[11] In iteration i, let $\pi_{\theta^{(i)}}$ be the current policy, and let $\pi_{\theta^{(i)\prime}}$ be the policy returned by DDP (applied to the MDP $\hat{M}^{(i+1)}$). Then, we have that $d^{(i)} = \theta^{(i)\prime} - \theta^{(i)}$ is the policy improvement direction. For the line search, we initialized $\alpha = 1$ (which corresponds to the policy $\pi_{\theta^{(i)\prime}}$), and then reduced α by 50% until an improved policy was found.

[12] We discarded approximate models for which the initial (model-based) controller was unstable in the real system. (This was $\pm$20% of the approximate models.)

[13] Details of the setup: The fixed-wing flight simulator experiments were produced using the linearized 6-DOF model developed in Chapter 2 of Stevens and Lewis (2003). The parameterization is intended to model a small (1-kilogram, 1.5-meter wingspan) wing-body UAV that flies at low airspeeds (10 m/s). The simulation ran at 50Hz. The target trajectory was a figure-8 at fixed altitude with varying speed: slower in the turns, faster in the straight segments. The penalties on the inputs and on the changes in inputs make the controller more robust to model inaccuracies. The trajectory was 60 seconds long.

[14] The (OpenGL) graphical flight simulator was originally developed by one of the authors, and is now maintained at http://sourceforge.net/projects/aviones.

Figure 2 shows a screenshot of the graphical simulator.

5.2. RC Car

In our second set of experiments, we used the RC car shown in Figure 3. A ceiling-mounted camera was used to obtain position estimates of the front and the rear of the car (marked with red and blue markers). This allowed us to track the car reliably within a rectangle of 3 by 2 meters. We then used an extended Kalman filter to track the car's state based on these observations in real-time.[15]

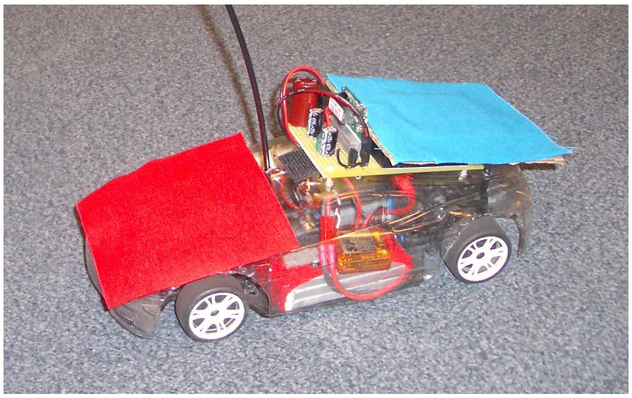

Figure 3. The RC car used in our experiments.

We modeled the RC car with six state-variables: the steering angle δ, the forward velocity of the front wheels v, the position coordinates x, y, the heading angle ψ and the heading rate $\dot{\psi}$. The inputs are the steering wheel servo input ($u_1 \in [-1, +1]$) and the drive motor throttle ($u_2 \in [-1, +1]$). At the speeds we consider, the slippage of front and rear wheels is negligible. Rigid-body kinematics then gives the heading rate $\dot{\psi}$ as a function of steering angle and velocity.[16] Data collected from the car showed that, when the throttle is decreased, the car slows down with fixed acceleration. However, when the throttle is increased, the response is similar to a first order linear model. This results in the following model:

$$\begin{aligned}
\delta_{\rm ss} &= a_1 u_1 + b_1, \\
\dot{\delta} &= (\delta_{\rm ss} - \delta)/\tau_\delta, \\
v_{\rm ss} &= a_2 u_2 + a_3 |u_1| + b_2, \\
\dot{v} &= \mathbf{1}\{v_{\rm ss} \geq v\}(v_{\rm ss} - v)/\tau_v + \mathbf{1}\{v_{\rm ss} < v\} a_4, \\
\dot{\psi} &= \frac{v \sin\delta}{L}.
\end{aligned}$$

Here $\mathbf{1}\{\cdot\}$ is the indicator function, which returns one if its argument is true and zero otherwise; $\delta_{\rm ss}$ is the steady state steering angle for fixed input u_1; $v_{\rm ss}$ is the steady-state velocity for fixed inputs u_1 and u_2. We find the state variables x, y, ψ by numerical integration. The parameters $(a_1, b_1, a_2, b_2, a_3, a_4, \tau_\delta, \tau_v, L)$ were estimated from measurements from the car and data collected from manually driving the car.[17]

We considered three control problems: a turn with an open-loop controller, circles with a closed-loop controller, and a figure-8 with a closed-loop controller. During the open-loop turn, the desired velocity and turning radius change. Thus, the control task requires a sequence of inputs, not just a single throttle and steering input setting. The circle maneuver is nontrivial because of the velocity (1.4m/s) and the tight turning radius. Moreover, the carpet threading makes the RC car drift, and thus the circle cannot be driven with a fixed throttle and steering input. The figure-8 maneuver is the most challenging of the three maneuvers: the car is required to drive at varying speeds (fast in straight segments (1.4m/s), slower in turns (1.0m/s)) and is required to make sharp turns, which requires fast and accurate control at the desired velocity. In each case, our reward function penalizes quadratically for deviation from the target trajectory (which defines a target state for each time step) and penalizes quadratically for inputs and changes in the

[15]Experimental setup details: The RC car is an off-the-shelf XRAY M18, which has an electric drive motor and an electric steering servo motor. The car was fitted with lower gearing to allow it to run reliably at lower speeds (necessary to stay within the 3 by 2 meters area that can be tracked by the camera). It was fitted with a switching voltage regulator to prevent battery decay from affecting performance. The video stream captured by the ceiling mounted camera was digitized by a 2GHz Linux workstation and processed by OpenCV (Intel, 2001) to dewarp the camera image. Custom software was then used to locate the largest centroid of red and blue in the image, and to map those image locations back to their respective positions in the ground plane. This experimental setup was inspired in part by a similar one in Andrew W. Moore's Auton Lab at Carnegie-Mellon University.

[16]See, e.g., Gillespie (1992) pp. 196-201, for details on low-speed turning and details on car modeling in general.

[17]We measured the wheelbase of the car L. We measured the steering angle for various inputs u_1, and used linear regression to initialize the parameters a_1, b_1. We collected about 30 minutes of data (at 30Hz) from the RC car driving in circles with different radii and at various speeds, including the velocities and turning radii for the control tasks to come. We used linear regression to estimate the parameters a_2, b_2, a_3 from the data segments where steady-state velocity was reached. We used step-input responses to initialize the parameters τ_δ, τ_v, a_4. We fine-tuned the parameters $a_1, b_1, \tau_\delta, \tau_v, a_4$ by maximizing the likelihood of the data (using coordinate ascent). The model included that when the throttle input was very close to 0 (more specifically, below 0.18), the RC car's drive motor would stall, i.e., the drive motor acted as if its input was 0. The model included the latency from inputs to camera observation: it was estimated from step-input responses to be 0.1 seconds. Modeled latency does not affect, and is orthogonal to, our method. However, unmodeled latency contributes to model error and could thus decrease performance.

inputs. The penalty terms on the inputs make the controller more robust to model inaccuracies. (See, e.g., Anderson and Moore (1989) for more details.)[18]

For all three control problems our algorithm significantly outperforms the model-based controller after 5-10 iterations, with typically 1-4 real-life executions required per iteration (due to the line search). Figure 4 shows the improvements throughout the algorithm for the case of the turn with open loop control. We see that the algorithm consistently makes progress to finally follow the turn almost perfectly. This is in sharp contrast to the performance of the model-based controller. Figure 5 shows the initial (model-based) trajectory and the trajectory obtained with our algorithm for the cases of closed-loop control for following a circle and a figure-8. Our algorithm improved the performance (utility) by 97%, 88% and 63% for the turn, circles and figure-8, respectively. Videos of the real-life trials when learning to drive the different trajectories are available at the url given previously.[19]

The sub-optimal performance of the model-based policy approach indicates model inaccuracies. Given the amount of training data collected (covering the velocities and turning radii required for the control tasks later), we believe that the model inaccuracies are caused by our car modeling assumptions, rather than lack of data. This suggests a more complex car model might be needed for the model-based approach. Additional model inaccuracy results from the carpet threading. The asymmetric carpet threading causes the RC car to drift by up to 0.10m per circle when driving circles of 1m radius in open-loop (fixed throttle and steering inputs). The carpet threading also seems to affect the stalling speed of the drive motor.

To evaluate how deterministic the RC car setup is, we repeatedly ran a few open-loop input sequences. For the same input sequence, the end-points of the trajectories differed from a few centimeters up to 50cm (for 2 second runs). We believe the (apparent) nondeterminism was caused mostly by the (unavoidably) slightly differing initial positions, which result in different interactions with the carpet throughout the trajectory. This effect is most significant when the car drives around the stalling speed of the drive motor.

[18] Control setup details: differential dynamic programming generated a sequence of linear feedback controllers (for the closed-loop case) and a sequence of inputs (for the open-loop case), all at 20Hz. The extended Kalman filter updated its state at roughly 30Hz (the frame rate of the camera). Whenever the state estimate was updated, the controller for the time closest to the current time was used. All three target trajectories started from zero velocity.

[19] We repeated the experiments 3-4 times for each maneuver, with similar results. The reported results correspond to the movies we put available online.

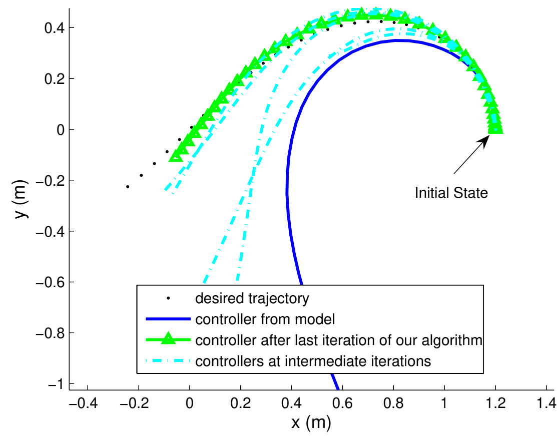

Figure 4. Results of our algorithm for the task of a turn with open-loop controller. (See text for details.)

6. Related Work

Classical and (especially) robust control theory consider the design of controllers that work well for a large set of systems that are similar to the model that is used for designing the controller. The resulting controllers are less vulnerable to modeling errors, but do not achieve optimal performance in the true system. (See, e.g., Zhou et al. (1995); Dullerud and Paganini (2000) for more details on robust control. See, e.g., Bagnell et al. (2001); Nilim and El Ghaoui (2005); Morimoto and Atkeson (2002) for some examples of robust-control work within the RL community.) Also within the formalism of optimal control one can give up optimality and, instead, design controllers that are more robust to mismatches between model and real-life, namely by including additional terms into the reward function. For example, one penalizes high frequency inputs (since the simulator is typically less accurate at high frequencies), or one penalizes the integral of the error over time to avoid steady-state errors. (See, e.g., Anderson and Moore (1989) for the linear quadratic setting.)

Another approach is to use non-parametric learning algorithms to build a model. (Atkeson et al., 1997) Although they might require more training data, non-parametric learning algorithms have the advantage of being less prone to modeling errors.

We note that Atkeson and Schaal (1997) and Morimoto and Doya (2001) also achieved successful robot control with a limited number of real-life trials. Our work significantly differs from both. Atkeson and Schaal (1997) started from a human demonstration of the (swing-up) task. Morimoto and Doya (2001) use a (hierarchical) Q-learning variant.

Iterative learning control (ILC) is the research area most closely related to our work, as our work could be

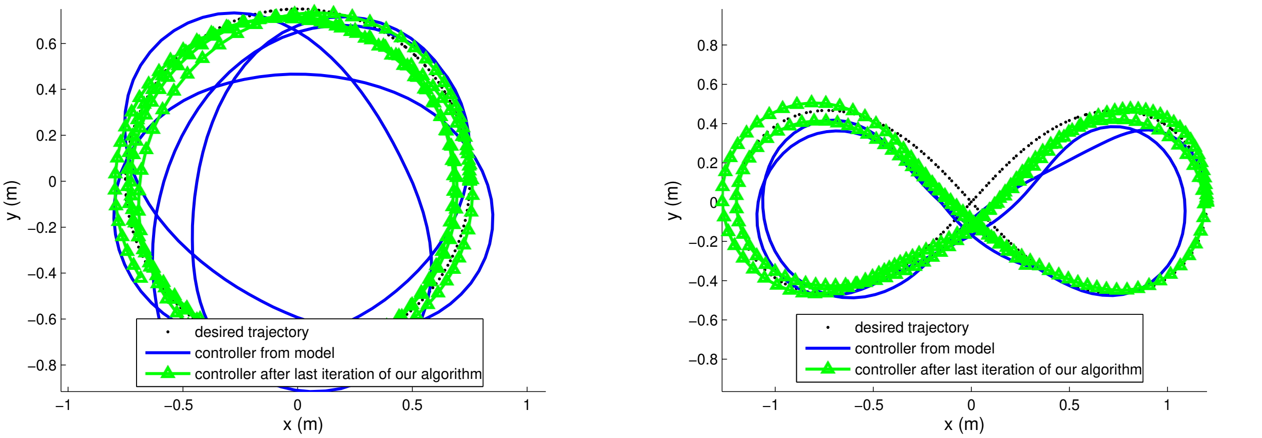

Figure 5. Results of our algorithm for the task of following a circle (a) and a figure-8 (b). (See text for details.)

seen as an instance of iterative learning control. ILC refers to the entire family of iterative control design approaches where the errors from past trials are used to compute the current inputs. In its most common form, ILC iteratively optimizes an open-loop (or feed-forward) input sequence. In contrast to our approach, most work in ILC does not use a model, rather, it just uses the past trials to compute the next iteration's controller. See, e.g., Moore (1998), for an overview and references to other work in iterative learning control.

7. Discussion

We presented an algorithm that uses a crude model and a small number of real-life trials to find a controller that works well in real-life. Our theoretical results show that—assuming the model derivatives are good approximations of the true derivatives, and assuming a deterministic setting—our algorithm will return a policy that is (locally) near-optimal. Our experiments (with a flight simulator and a real RC car) demonstrate that our algorithm can significantly outperform the model-based control approach, even when using only an approximate model and a few real-life trials, and when the true system is not (perfectly) deterministic.

An interesting future research direction is to extend our algorithm to the case of stochastic dynamics.

Acknowledgments

We give warm thanks to Mark Woodward for developing the RC car tracking software, and to Chris Atkeson for his helpful comments. M. Quigley is supported by a DoD NDSEG fellowship. This work was supported in part by the DARPA Learning Locomotion program under contract number FA8650-05-C-7261.

References

Abbeel, P., Quigley, M., & Ng, A. Y. (2006). Using inaccurate models in reinforcement learning. (Full paper.) `http://www.cs.stanford.edu/~pabbeel/`.

Anderson, B., & Moore, J. (1989). *Optimal control: Linear quadratic methods*. Prentice-Hall.

Atkeson, C. G., Moore, A. W., & Schaal, S. (1997). Locally weighted learning for control. *AI Review*.

Atkeson, C. G., & Schaal, S. (1997). Robot learning from demonstration. *Proc. ICML*.

Bagnell, J., Ng, A. Y., & Schneider, J. (2001). *Solving uncertain Markov decision problems* (Technical Report). Robotics Institute, Carnegie-Mellon University.

Bertsekas, D. P. (2001). *Dynamic programming and optimal control*, vol. 1. Athena Scientific. 2nd edition.

Bertsekas, D. P., & Tsitsiklis, J. (1996). *Neuro-dynamic programming*. Athena Scientific.

Dullerud, G. E., & Paganini, F. (2000). *A course in robust control theory: A convex approach*, vol. 36 of *Texts in Applied Mathematics*. Springer - New York.

Gillespie, T. (1992). *Fundamentals of vehicle dynamics*. SAE.

Intel (2001). Opencv libraries for computer vision. http://www.intel.com/research/mrl/research/opencv/.

Jacobson, D. H., & Mayne, D. Q. (1970). *Differential dynamic programming*. Elsevier.

Kohl, N., & Stone, P. (2004). Machine learning for fast quadrupedal locomotion. *Proc. AAAI*.

Moore, K. L. (1998). Iterative learning control: An expository overview. *Applied and Computational Controls, Signal Processing, and Circuits*.

Morimoto, J., & Atkeson, C. G. (2002). Minimax differential dynamic programming: An application to robust biped walking. *NIPS 14*.

Morimoto, J., & Doya, K. (2001). Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. *Robotics and Autonomous Systems*.

Nilim, A., & El Ghaoui, L. (2005). Robust solutions to Markov decision problems with uncertain transition matrices. *Operations Research*.

Stevens, B. L., & Lewis, F. L. (2003). *Aircraft control and simulation*. Wiley and Sons. 2nd edition.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT Press.

Zhou, K., Doyle, J., & Glover, K. (1995). *Robust and optimal control*. Prentice Hall.