Direct Gradient-Based Reinforcement Learning: II. Gradient Ascent Algorithms and Experiments

Jonathan Baxter
Research School of Information Sciences and Engineering
Australian National University
Jonathan.Baxter@anu.edu.au

Lex Weaver
Department of Computer Science
Australian National University
Lex.Weaver@anu.edu.au

Peter Bartlett
Research School of Information Sciences and Engineering
Australian National University
Peter.Bartlett@anu.edu.au

September 20, 1999
Revised: November 27, 1999

November 27, 1999

Abstract

In [2] we introduced GPOMDP, an algorithm for computing arbitrarily accurate approximations to the performance gradient of parameterized partially observable Markov decision processes (POMDPs).

The algorithm's chief advantages are that it requires only a single sample path of the underlying Markov chain, it uses only one free parameter $\beta \in [0, 1)$ which has a natural interpretation in terms of bias-variance trade-off, and it requires no knowledge of the underlying state. In addition, the algorithm can be applied to infinite state, control and observation spaces.

In this paper we present CONJPOMDP, a conjugate-gradient ascent algorithm that uses GPOMDP as a subroutine to estimate the gradient direction. CONJPOMDP uses a novel line-search routine that relies solely on gradient estimates and hence is robust to noise in the performance estimates. OLPOMDP, an on-line gradient ascent algorithm based on GPOMDP is also presented.

The chief theoretical advantage of this gradient based approach over value-function-based approaches to reinforcement learning is that it guarantees improvement in the performance of the policy at *every* step. To show that this advantage

is real, we give experimental results in which CONJPOMDP was used to optimize a simple three-state Markov chain controlled by a linear function, a two-dimensional "puck" controlled by a neural network, a call admission queueing problem, and a variation of the classical "mountain-car" task. In all cases the algorithm rapidly found optimal or near-optimal solutions.

1 Introduction

Function approximation is necessary to avoid the curse of dimensionality associated with large-scale dynamic programming and reinforcement learning problems. The dominant paradigm is to use the function to approximate the state (or state and action) values. Most algorithms then seek to minimize some form of error between the approximate value function and the true value function, usually by simulation (see [13] and [4] for comprehensive overviews). While there have been a multitude of empirical successes for this approach (see e.g [10, 14, 15, 3, 18, 11] to name but a few), there are only weak theoretical guarantees on the performance of the policy generated by the approximate value function. In particular, there is no guarantee that the policy will improve as the approximate value function is trained (see [2, Section 1] for further discussion).

Motivated by these difficulties, in [2] we introduced GPOMDP, a new algorithm for computing arbitrarily accurate approximations to the performance gradient of parameterized partially observable Markov decision processes (POMDP's). Our algorithm is essentially an extension of Williams' REINFORCE algorithm [17] and similar more recent algorithms [7, 5, 9, 8]. See [2, Section 1.1] for a more comprehensive discussion of this related work.

More specifically, suppose $\theta \in \mathbb{R}^K$ are the parameters controlling the POMDP. For example, θ could be the parameters of an approximate neural-network value-function that generates a stochastic policy by some form of randomized look-ahead, or θ could be the parameters of an approximate Q function used to stochastically select controls[1]. Let $\eta(\theta)$ denote the average reward of the POMDP with parameter setting θ. GPOMDP computes an approximation $\nabla_\beta \eta(\theta)$ to $\nabla \eta(\theta)$ based on a single continuous sample path of the underlying Markov chain. The accuracy of the approximation is controlled by the parameter $\beta \in [0, 1)$. It was proved in [2, Theorem 3] that

$$\nabla \eta(\theta) = \lim_{\beta \to 1} \nabla_\beta \eta(\theta).$$

The trade-off preventing us choosing β arbitrarily close to 1 is that the variance of GPOMDP's estimates of $\nabla_\beta \eta(\theta)$ increase with β. However, on the bright side, [2, Theorem 4] showed that the approximation error is proportional to

$$\frac{1-\beta}{1-|\lambda_2|},$$

[1] Stochastic policies are not strictly necessary in our framework, but the policy must be "differentiable" in the sense that $\nabla \eta(\theta)$ exists.

where λ_2 is the subdominant eigenvalue of the Markov chain underlying the POMDP. Thus for "rapidly mixing" POMDP's (for which λ_2 is significantly less than 1), estimates of the performance gradient with acceptable *bias* and *variance* can be obtained.

Provided $\nabla_\beta \eta(\theta)$ is a sufficiently accurate approximation of $\nabla \eta(\theta)$—in fact, $\nabla_\beta \eta(\theta)$ need only be within 90° of $\nabla \eta(\theta)$—adjustments to the parameters θ of the form $\theta \leftarrow \theta + \gamma \nabla_\beta \eta(\theta)$ for small step-size γ, will guarantee improvement in the average reward $\eta(\theta)$. In this case, gradient-based optimization algorithms using $\nabla_\beta \eta(\theta)$ as their gradient estimate will be guaranteed to improve the average reward $\eta(\theta)$ on each step. Except in the case of table-lookup, most value-function based approaches to reinforcement learning cannot make this guarantee. See [16] for some analysis in the case of TD(λ) and a demonstration of performance degradation during the course of training a neural network backgammon player.

In this paper we present CONJPOMDP, a conjugate-gradient ascent algorithm that uses the estimates of $\nabla_\beta \eta(\theta)$ provided by GPOMDP. Critical to the successful operation of CONJPOMDP is a novel line search subroutine that reduces noise by relying solely upon gradient estimates. We also present OLPOMDP, an on-line variant of our algorithm that updates the parameters at every time step. OLPOMDP is similar to algorithms proposed in [7] and [9].

The two algorithms are applied to a variety of problems, beginning with a simple 3-state Markov decision process (MDP) controlled by a linear function for which the true gradient can be exactly computed. We show rapid convergence of the gradient estimates $\nabla_\beta \eta(\theta)$ to the true gradient, in this case over a large range of values of β. With this simple system we are able to illustrate vividly the bias/variance tradeoff associated with the selection of β. We then use CONJPOMDP and OLPOMDP to find a good policy for the MDP. CONJPOMDP reliably finds a near-optimal policy in less than 100 iterations of the Markov chain, an order of magnitude faster than OLPOMDP.

Next we demonstrate the effectiveness of CONJPOMDP in training a neural network controller to control a "puck" in a two-dimensional world. The task in this case is to reliably navigate the puck from any starting configuration to an arbitrary target location in the minimum time, while only applying discrete forces in the x and y directions.

In the third experiment, we use CONJPOMDP to train a controller for the call admission queueing problem treated in [8]. In this case CONJPOMDP finds near-optimal solutions within about 2000 iterations of the underlying queue.

In the fourth and final experiment, CONJPOMDP is used to train a switched neural-network controller for a two-dimensional variation on the classical "mountain-car" task [13, Example 8.2].

The rest of this paper is organized as follows. In Section 2 we introduce the definitions needed to understand GPOMDP. In Section 3 we describe CONJPOMDP, the gradient-based line-search subroutine, and OLPOMDP. In Section 4 we present our experimental results.

2 The GPOMDP algorithm

A partially observable, Markov decision process (POMDP) consists of a state space $\mathcal{S}$, observation space $\mathcal{Y}$ and a control space $\mathcal{U}$. For each state $i \in \mathcal{S}$ there is a deterministic reward $r(i)$. Although the results in [2] only guarantee convergence of GPOMDP in the case of finite $\mathcal{S}$ (but rather arbitrary $\mathcal{U}$ and $\mathcal{Y}$), the algorithm can be applied regardless of the nature of $\mathcal{S}$ so we do not restrict the cardinality of $\mathcal{S}$, $\mathcal{U}$ or $\mathcal{Y}$.

Consider first the case of discrete $\mathcal{S}$, $\mathcal{U}$ and $\mathcal{Y}$. Each control $u \in \mathcal{U}$ determines a stochastic matrix $P(u) = [p_{ij}(u)]$ giving the transition probability from state i to state j ($i, j \in \mathcal{S}$). For each state $i \in \mathcal{S}$, an observation $y \in \mathcal{Y}$ is generated independently according to a probability distribution $\nu(i)$ over observations in $\mathcal{Y}$. We denote the probability of y by $\nu_y(i)$. A *randomized policy* is simply a function μ mapping observations into probability distributions over the controls $\mathcal{U}$. That is, for each observation $y \in \mathcal{Y}$, $\mu(y)$ is a distribution over the controls in $\mathcal{U}$. Denote the probability under μ of control u given observation y by $\mu_u(y)$.

For continuous $\mathcal{S}, \mathcal{Y}$ and $\mathcal{U}$, $p_{ij}(u)$ becomes a *kernel* $k_{ij}(u)$ giving the probability density of transitions from i to j, $\nu(i)$ becomes a probability density function on $\mathcal{Y}$ with $\nu_y(i)$ the density at y, and $\mu(y)$ becomes a probability density function on $\mathcal{U}$ with $\mu_u(y)$ the density at u.

To each randomized policy μ there corresponds a Markov chain in which state transitions are generated by first selecting an observation y in state i according to the distribution $\nu(i)$, then selecting a control u according to the distribution $\mu(y)$, and finally generating a transition to state j according to the probability $p_{ij}(u)$.

At present we are only dealing with a fixed POMDP. To parameterize the POMDP we parameterize the policies, so that μ now becomes a function $\mu(\theta, y)$ of a set of parameters $\theta \in \mathbb{R}^K$, as well as of the observation y. The Markov chain corresponding to θ has state transition matrix $P(\theta) = [p_{ij}(\theta)]$ given by

$$p_{ij}(\theta) = \mathbf{E}_{y \sim \nu(i)} \mathbf{E}_{u \sim \mu(\theta, y)} p_{ij}(u). \tag{1}$$

The following technical assumptions are required for the operation of GPOMDP.

Assumption 1. *The derivatives,*

$$\left[\frac{\partial \mu_u(\theta, y)}{\partial \theta_k} \right]_{k=1 \ldots K}$$

exist for all $u \in \mathcal{U}$, $y \in \mathcal{Y}$ and $\theta \in \mathbb{R}^K$.

Assumption 2. *The ratios*

$$\left[\frac{\left| \frac{\partial \mu_u(\theta, y)}{\partial \theta_k} \right|}{\mu_u(\theta, y)} \right]_{k=1 \ldots K}$$

are uniformly bounded by $B < \infty$, for all $u \in \mathcal{U}$, $y \in \mathcal{Y}$ and $\theta \in \mathbb{R}^K$.

Assumption 3. *The magnitudes of the rewards, $|r(i)|$, are uniformly bounded by $R < \infty$ for all states i.*

Assumption 4. *Each $P(\theta), \theta \in \mathbb{R}^K$, has a unique stationary distribution, $\pi(\theta)$.*

The *average reward* $\eta(\theta)$ is simply the expected reward under the stationary distribution $\pi(\theta)$:

$$\eta(\theta) = \mathbf{E}_{i \sim \pi(\theta)} r(i). \tag{2}$$

Because of Assumption 4, for any starting state i, $\eta(\theta)$ is also equal to the expected long-term average of the reward,

$$\lim_{T \to \infty} \mathbf{E} \left(\frac{1}{T} \sum_{i=0}^{T-1} r(i_t) \middle| i_0 = i \right),$$

where the expectation is over sequences of states $i_0, \ldots, i_{T-1}$ of the Markov chain specified by $P(\theta)$.

GPOMDP ([2, Algorithm 2] and reproduced in Algorithm 1) is an algorithm for computing an approximation Δ_T to $\nabla \eta(\theta)$. In [2, Theorem 7] we proved:

$$\lim_{T \to \infty} \Delta_T = \nabla_\beta \eta(\theta),$$

where $\nabla_\beta \eta(\theta)$ ($\beta \in [0, 1)$) is an approximation to $\nabla \eta(\theta)$ satisfying

$$\nabla \eta(\theta) = \lim_{\beta \to 1} \nabla_\beta \eta(\theta),$$

[2, Theorem 3]. Note that GPOMDP relies only upon a single sample path from the POMDP. Also, it does not require knowledge of the transition probability matrix P, nor of the observation process ν; it only requires knowledge of the randomized policy μ.

We cannot set β arbitrarily close to 1 in GPOMDP, since the variance of the estimate Δ_T increases with increasing β. Thus β has a natural interpretation in terms of a bias-variance trade-off: small values of β give lower variance in the estimates Δ_T, but higher bias in that Δ_T may be far from $\nabla \eta(\theta)$, whereas values of β close to 1 yield small bias but correspondingly larger variance. This bias/variance trade-off is vividly illustrated in the experiments of Section 4.

3 Stochastic gradient ascent algorithms

In this section we introduce two algorithms: CONJPOMDP, a variant of the Polak-Ribiere conjugate gradient algorithm (see e.g. [6, S5.5.2]), and OLPOMDP, a fully on-line algorithm that updates the parameters θ at each iteration of the POMDP.

3.1 The CONJPOMDP algorithm

CONJPOMDP, described in Algorithm 2, is a version of the Polak-Ribiere conjugate-gradient algorithm that is designed to operate using only noisy (and possibly) biased

Algorithm 1 $\mathrm{GPOMDP}(\beta, T, \theta) \to \mathbb{R}^K$ [2, Algorithm 2].

Given:

- $\beta \in [0, 1)$.
- $T > 0$.
- Parameters $\theta \in \mathbb{R}^K$.
- Randomized policy $\mu(\theta, \cdot)$ satisfying Assumptions 1 and 2.
- POMDP with rewards satisfying Assumption 3, and which when controlled by $\mu(\theta, \cdot)$ generates stochastic matrices $P(\theta)$ satisfying Assumption 4.
- Arbitrary (unknown) starting state i_0.

Set $z_0 = 0$ and $\Delta_0 = 0$ ($z_0, \Delta_0 \in \mathbb{R}^K$).
for $t = 0$ to $T - 1$ **do**
    Observe y_t (generated according to $\nu(i_t)$)
    Generate control u_t according to $\mu(\theta, y_t)$
    Observe $r(i_{t+1})$ (where the next state i_{t+1} is generated according to $p_{i_t i_{t+1}}(u_t)$).
    Set $z_{t+1} = \beta z_t + \frac{\nabla \mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)}$
    Set $\Delta_{t+1} = \Delta_t + r(i_{t+1}) z_{t+1}$
end for
$\Delta_T \leftarrow \Delta_T / T$
return Δ_T

estimates of the gradient of the objective function (for example, the estimates Δ_T provided by GPOMDP). The novel feature of CONJPOMDP is GSEARCH, a linesearch subroutine that uses only gradient information to find the local maximum in the search direction. The use of gradient information ensures GSEARCH is robust to noise in the performance estimates. Both CONJPOMDP and GSEARCH can be applied to any stochastic optimization problem for which noisy (and possibly) biased gradient estimates are available.

The argument s_0 to CONJPOMDP provides an initial step-size for GSEARCH. When $\|\mathrm{GRAD}(\theta)\|^2$ falls below the argument ϵ, CONJPOMDP terminates.

Algorithm 2 CONJPOMDP$(\mathrm{GRAD}, \theta, s_0, \epsilon)$

Given:

- $\mathrm{GRAD}\colon \mathbb{R}^K \to \mathbb{R}^K$: a (possibly noisy and biased) estimate of the gradient of the objective function to be maximized.
- Starting parameters $\theta \in \mathbb{R}^K$ (set to maximum on return).
- Initial step size $s_0 > 0$.
- Gradient resolution ϵ.

$g = h = \mathrm{GRAD}(\theta)$
while $\|g\|^2 \geq \epsilon$ **do**
  GSEARCH$(\mathrm{GRAD}, \theta, h, s_0, \epsilon)$
  $\Delta = \mathrm{GRAD}(\theta)$
  $\gamma = (\Delta - g) \cdot \Delta / \|g\|^2$
  $h = \Delta + \gamma h$
  if $h \cdot \Delta < 0$ **then**
    $h = \Delta$
  end if
  $g = \Delta$
end while

3.2 The GSEARCH algorithm

The key to the successful operation of CONJPOMDP is the linesearch algorithm GSEARCH (Algorithm 3). GSEARCH uses only gradient information to bracket the maximum in the direction θ^*, and then quadratic interpolation to jump to the maximum.

We found the use of gradients to bracket the maximum far more robust than the use of function values. To bracket the maximum using function values, three points $\theta_1, \theta_2, \theta_3$, all lying in the direction θ^* from θ, must be found such that $\eta(\theta_1) < \eta(\theta_2)$ and $\eta(\theta_3) < \eta(\theta_2)$. Thus, we need to estimate $\mathrm{sign}[\eta(\theta_1) - \eta(\theta_2)]$ (and $\mathrm{sign}[\eta(\theta_3) - \eta(\theta_2)]$). If we only have access to noisy estimates of $\eta(\theta)$ (for example, estimates obtained by simulation), then regardless of the magnitude of the variance of $\eta(\theta)$, the variance of $\mathrm{sign}\,[\eta(\theta_1) - \eta(\theta_2)]$ approaches 1 (the maximum possible) as θ_1

approaches θ_2. Thus, to reliably bracket the maximum using noisy estimates of $\eta(\theta)$ we need to be able to reduce the variance of the estimates when θ_1 and θ_2 are close. In our case this means running the simulation from which the estimates are derived for longer and longer periods of time.

An alternative approach to bracketing the maximum in the direction θ^* from θ is to find two points θ_1 and θ_2 in that direction such that $\mathrm{GRAD}(\theta_1) \cdot \theta^* > 0$ and $\mathrm{GRAD}(\theta_2) \cdot \theta^* < 0$. The maximum must then lie between θ_1 and θ_2. The advantage of this approach is that even if the estimates $\mathrm{GRAD}(\theta)$ are noisy, the variance of $\mathrm{sign}\,[\mathrm{GRAD}(\theta_1) \cdot \theta^*]$ (and $\mathrm{sign}\,[\mathrm{GRAD}(\theta_2) \cdot \theta^*]$) is independent of the distance between θ_1 and θ_2, and in particular does not grow as the two points approach one another. The disadvantage is that it is not possible to detect extreme overshooting of the maximum using only gradient estimates. However, with careful control of the line search we did not find this to be a problem.

In Algorithm 3, lines 5–25 bracket the maximum by finding a parameter setting $\theta_- = \theta_0 + s_-\theta^*$ such that $\mathrm{GRAD}(\theta_-) \cdot \theta^* > -\epsilon$, and a second parameter setting $\theta_+ = \theta_0 + s_+\theta^*$ such that $\mathrm{GRAD}(\theta_+) \cdot \theta^* < \epsilon$. The reason for ϵ rather than 0 in these expressions is to provide some robustness against errors in the estimates $\mathrm{GRAD}(\theta)$. It also prevents the algorithm "stepping to ∞" if there is no local maximum in the direction θ^*. Note that we use the same ϵ as used in CONJPOMDP to determine when to terminate due to small gradient (line 4 in CONJPOMDP).

Provided that the signs of the gradients at the bracketing points θ_- and θ_+ show that the maximum of the quadratic defined by these points lies between them, line 27 will jump to the maximum. Otherwise the algorithm simply jumps to the midpoint between θ_- and θ_+.

3.3 OLPOMDP: updating the parameters θ at every time step

CONJPOMDP operates by iteratively choosing "uphill" directions and then searching for a local maximum in the chosen direction. If the GRAD argument to CONJPOMDP is GPOMDP, the optimization will involve many iterations of the underlying POMDP between parameter updates.

An alternative approach, similar in spirit to algorithms described in [7, 9, 8], is to adjust the parameter vector at every iteration of the underlying POMDP. Algorithm 4, OLPOMDP, presents one such algorithm along these lines. We are currently working on a convergence proof for this algorithm.

4 Experiments

In this section we present several sets of experimental results. Throughout this section, where we refer to CONJPOMDP we mean CONJPOMDP with GPOMDP as its GRAD argument.

In the first set of experiments, we consider a system in which a controller is used to select actions for a 3-state Markov Decision Process (MDP). For this system we are

Algorithm 3 $\mathrm{GSEARCH}(\mathrm{GRAD}, \theta_0, \theta^*, s_0, \epsilon)$

Given:

- $\mathrm{GRAD}\colon \mathbb{R}^K \to \mathbb{R}^K$: a (possibly noisy and biased) estimate of the gradient of the objective function.
- Starting parameters $\theta_0 \in \mathbb{R}^K$ (set to maximum on return).
- Search direction $\theta^* \in \mathbb{R}^K$ with $\mathrm{GRAD}(\theta_0) \cdot \theta^* > 0$.
- Initial step size $s_0 > 0$.
- Inner product resolution $\epsilon >= 0$.

$s = s_0$
$\theta = \theta_0 + s\theta^*$
$\Delta = \mathrm{GRAD}(\theta)$
if $\Delta \cdot \theta^* < 0$ **then**
  Step back to bracket the maximum:
  repeat
    $s_+ = s$
    $p_+ = \Delta \cdot \theta^*$
    $s = s/2$
    $\theta = \theta_0 + s\theta^*$
    $\Delta = \mathrm{GRAD}(\theta)$
  until $\Delta \cdot \theta^* > -\epsilon$
  $s_- = s$
  $p_- = \Delta \cdot \theta^*$
else
  Step forward to bracket the maximum:
  repeat
    $s_- = s$
    $p_- = \Delta \cdot \theta^*$
    $s = 2s$
    $\theta = \theta_0 + s\theta^*$
    $\Delta = \mathrm{GRAD}(\theta)$
  until $\Delta \cdot \theta^* < \epsilon$
  $s_+ = s$
  $p_+ = \Delta \cdot \theta^*$
end if
if $p_- > 0$ and $p_+ < 0$ **then**
  $s = \frac{s_- p_+ - s_+ p_-}{p_+ - p_-}$
else
  $s = \frac{s_- + s_+}{2}$
end if
$\theta_0 = \theta_0 + s\theta^*$

Algorithm 4 $\mathrm{OLPOMDP}(\beta, T, \theta_0) \to \mathbb{R}^K$.

Given:

- $\beta \in [0, 1)$.
- $T > 0$.
- Initial parameter values $\theta_0 \in \mathbb{R}^K$.
- Randomized parameterized policies $\{\mu(\theta, \cdot) \colon \theta \in \mathbb{R}^K\}$ satisfying Assumptions 1 and 2.
- POMDP with rewards satisfying Assumption 3, and which when controlled by $\mu(\theta, \cdot)$ generates stochastic matrices $P(\theta)$ satisfying Assumption 4.
- Step sizes $\gamma_t, t = 0, 1, \ldots$ satisfying $\sum \gamma_t = \infty$ and $\sum \gamma_t^2 < \infty$.
- Arbitrary (unknown) starting state i_0.

Set $z_0 = 0$ ($z_0 \in \mathbb{R}^K$).
for $t = 0$ to $T - 1$ **do**
  Observe y_t (generated according to $\nu(i_t)$).
  Generate control u_t according to $\mu(\theta, y_t)$
  Observe $r(i_{t+1})$ (where the next state i_{t+1} is generated according to $p_{i_t i_{t+1}}(u_t)$.
  Set $z_{t+1} = \beta z_t + \frac{\nabla \mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)}$
  Set $\theta_{t+1} = \theta_t + \gamma_t r(i_{t+1}) z_{t+1}$
end for
return θ_T

able to compute the true gradient exactly using the matrix equation

$$\nabla\eta(\theta) = \pi'(\theta)\nabla P(\theta)\left[I - P(\theta) + e\pi'(\theta)\right]^{-1} r, \tag{3}$$

where $P(\theta)$ is the transition matrix of the underlying Markov chain with the controller's parameters set to θ, $\pi'(\theta)$ is the stationary distribution corresponding to $P(\theta)$ (written as a row vector), $e\pi'(\theta)$ is the matrix in which each row is the stationary distribution, and r is the (column) vector of rewards (see [2, S2.1] for a derivation of (3)). Hence we can compare the estimates Δ_T generated by GPOMDP with the true gradient $\nabla\eta(\theta)$, both as a function of the number of iterations T and as a function of the discount parameter β. We also optimize the performance of the controller using the on-line algorithm, OLPOMDP, and CONJPOMDP. CONJPOMDP reliably converges to a near optimal policy with around 100 iterations of the MDP, while the on-line method requires approximately 1000 iterations. This should be contrasted with training a linear *value-function* for this system using TD(1) [12], which can be shown to converge to a value function whose one-step lookahead policy is suboptimal [16].

In the second set of experiments, we consider a simple "puck-world" problem in which a small puck must be navigated around a two-dimensional world by applying thrust in the x and y directions. We train a 1-hidden-layer neural-network controller for the puck using CONJPOMDP. Again the controller reliably converges to near optimality.

In the third set of experiments we use CONJPOMDP to optimize the admission thresholds for the call-admission problem considered in [8].

In the final set of experiments we use CONJPOMDP to train a switched neural-network controller for a two-dimensional variant of the "mountain-car" task [13, Example 8.2].

4.1 A three-state MDP

In this section we consider a three-state MDP, in each state of which there is a choice of two actions a_1 and a_2. Table 1 shows the transition probabilities as a function of the states and actions. Each state x has an associated two-dimensional feature vector $\phi(x) = (\phi_1(x), \phi_2(x))$ and reward $r(x)$ which are detailed in Table 2. Clearly, the optimal policy is to always select the action that leads to state C with the highest probability, which from Table 1 means always selecting action a_2.

This rather odd choice of feature vectors for the states ensures that a value function linear in those features and trained using TD(1)—while observing the optimal policy—will implement a suboptimal one-step greedy lookahead policy itself (see [16] for a proof). Thus, in contrast to the gradient based approach, for this system, TD(1) training a linear value function is guaranteed to produce a worse policy if it starts out observing the optimal policy.

4.1.1 Training a controller

Our goal is to learn a stochastic controller for this system that implements an optimal (or near-optimal) policy. Given a parameter vector $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$, we generate a

Origin State	Action	Destination State Probabilities A	B	C
A	$a1$	0.0	0.8	0.2
A	$a2$	0.0	0.2	0.8
B	$a1$	0.8	0.0	0.2
B	$a2$	0.2	0.0	0.8
C	$a1$	0.0	0.8	0.2
C	$a2$	0.0	0.2	0.8

Table 1: Transition probabilities of the three-state MDP

$r(A) = 0$	$\phi_1(A) = \frac{12}{18}$	$\phi_2(A) = \frac{6}{18}$
$r(B) = 0$	$\phi_1(B) = \frac{6}{18}$	$\phi_2(B) = \frac{12}{18}$
$r(C) = 1$	$\phi_1(C) = \frac{5}{18}$	$\phi_2(C) = \frac{5}{18}$

Table 2: Three-state rewards and features.

policy as follows. For any state x, let

$$s_1(x) := \theta_1\phi_1(x) + \theta_2\phi_2(x)$$
$$s_2(x) := \theta_3\phi_1(x) + \theta_4\phi_2(x).$$

Then the probability of choosing action a_1 in state x is given by

$$\mu_{a_1}(x) = \frac{e^{s_1(x)}}{e^{s_1(x)} + e^{s_2(x)}},$$

while the probability of choosing action a_2 is given by

$$\mu_{a_2}(x) = \frac{e^{s_2(x)}}{e^{s_1(x)} + e^{s_2(x)}} = 1 - \mu_{a_1}(x).$$

The ratios $\frac{\nabla\mu_{a_i}(x)}{\mu_{a_i}(x)}$ needed by Algorithms 1 and 4 are given by,

$$\frac{\nabla\mu_{a_1}(x)}{\mu_{a_1}(x)} = \frac{e^{s_2(x)}}{e^{s_1(x)} + e^{s_2(x)}}\left[\phi_1(x), \phi_2(x), -\phi_1(x), -\phi_2(x)\right] \tag{4}$$

$$\frac{\nabla\mu_{a_2}(x)}{\mu_{a_2}(x)} = \frac{e^{s_1(x)}}{e^{s_1(x)} + e^{s_2(x)}}\left[-\phi_1(x), -\phi_2(x), \phi_1(x), \phi_2(x)\right] \tag{5}$$

4.1.2 Gradient estimates

With a parameter vector[2] of $\theta = [1, 1, -1, -1]$, estimates Δ_T of $\nabla_\beta\eta$ were generated using GPOMDP, for various values of T and $\beta \in [0, 1)$. To measure the progress of

[2]Other initial values of the parameter vector were chosen with similar results. Note that $[1, 1, -1, -1]$ generates a suboptimal policy.

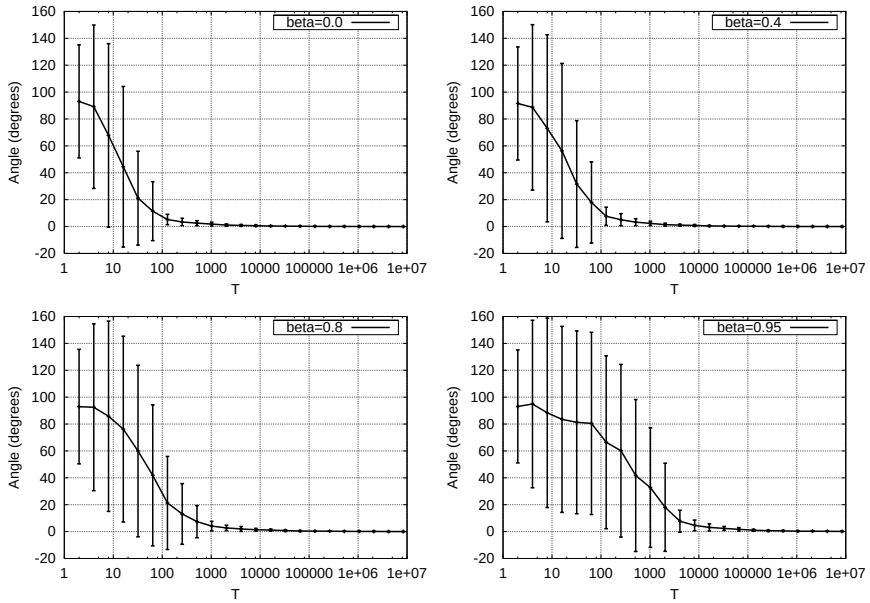

Figure 1: Angle between the true gradient $\nabla\eta$ and the estimate Δ_T for the three-state Markov chain, for various values of the discount parameter β. Δ_T was generated by Algorithm 1. Averaged over 500 independent runs. Note the higher variance at large T for the larger values of β. Error bars are one standard deviation.

Δ_T towards to the true gradient $\nabla\eta$, $\nabla\eta$ was calculated from (3) and then for each value of T the *angle* between Δ_T and $\nabla\eta$ and the relative error $\frac{\|\Delta_T - \nabla\eta\|}{\|\nabla\eta\|}$ were recorded. The angles and relative errors are plotted in Figures 1, 2 and 3.

The graphs illustrate a typical trade-off for the GPOMDP algorithm: small values of β give higher bias in the estimates, while larger values of β give higher variance (the bias is only shown in Figure 3 for the norm deviation because it was too small to measure for the angular deviation). That said, the bias introduced by having $\beta < 1$ is very small for this system. In the worst case, $\beta = 0.0$, the final gradient *direction* is indistinguishable from the true direction while the relative deviation $\frac{\|\nabla\eta - \Delta_T\|}{\|\nabla\eta\|}$ is only 7.7%.

4.1.3 Training via conjugate-gradient ascent

CONJPOMDP with GPOMDP as the "GRAD" argument was used to train the parameters of the controller described in the previous section. Following the low bias observed in the experiments of the previous section, the argument β of GPOMDP was set to 0. After a small amount of experimentation, the arguments s_0 and ϵ of CONJPOMDP were set to 100 and 0.0001 respectively. None of these values were critical, although the extremely large initial step-size (s_0) did considerably reduce the time required for

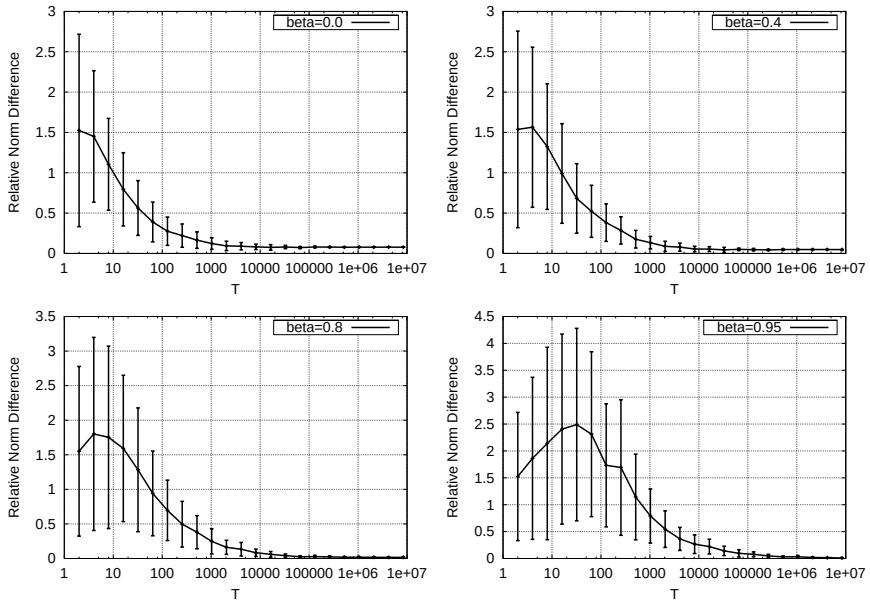

Figure 2: A plot of $\frac{\|\nabla\eta - \Delta_T\|}{\|\nabla\eta\|}$ for the three-state Markov chain, for various values of the discount parameter β. Δ_T was generated by Algorithm 1. Averaged over 500 independent runs. Note the higher variance at large T for the larger values of β. Error bars are one standard deviation.

the controller to converge to near-optimality.

We tested the performance of CONJPOMDP for a range of values of the argument T to GPOMDP from 1 to 4096. Since GSEARCH only uses GPOMDP to determine the *sign* of the inner product of the gradient with the search direction, it does not need to run GPOMDP for as many iterations as CONJPOMDP does. Thus, GSEARCH determined its own T parameter to GPOMDP as follows. Initially, (somewhat arbitrarily) the value of T within GSEARCH was set to $1/10$ the value used in CONJPOMDP (or 1 if the value in CONJPOMDP was less than 10). GSEARCH then called GPOMDP to obtain an estimate Δ_T of the gradient direction. If $\Delta_T \cdot \theta^* < 0$ (θ^* being the desired search direction) then T was doubled and GSEARCH was called again to generate a new estimate Δ_T. This procedure was repeated until $\Delta_T \cdot \theta^* > 0$, or T had been doubled four times. If $\Delta_T \cdot \theta^*$ was still negative at the end of this process, GSEARCH searched for a local maximum in the direction $-\theta^*$, and the number of iterations T used by CONJPOMDP was doubled on the next iteration (the conclusion being that the direction θ^* was generated by overly noisy estimates from GPOMDP).

Figure 4 shows the average reward $\eta(\theta)$ of the final controller produced by CONJPOMDP, as a function of the total number of simulation steps of the underlying Markov chain. The plots represent an average over 500 independent runs of CONJPOMDP. Note that 0.8 is the average reward of the optimal policy. The param-

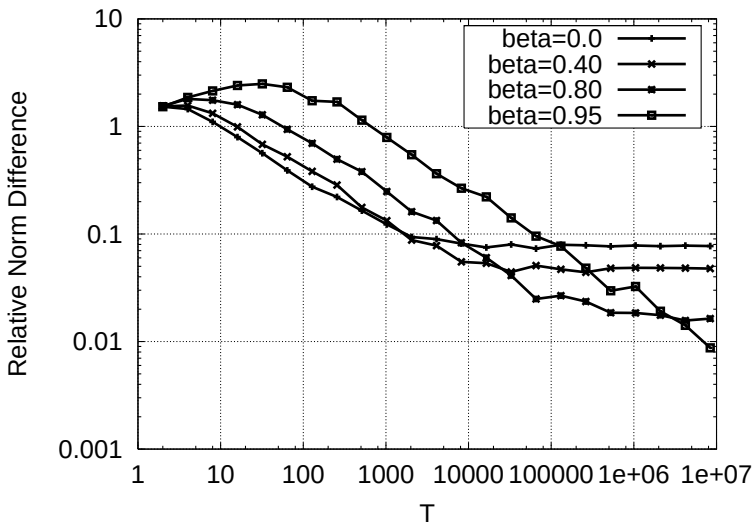

Figure 3: Graph showing the final bias in the estimate Δ_T (as measured by $\frac{\|\nabla\eta - \Delta_T\|}{\|\nabla\eta\|}$) as a function of β for the three-state Markov chain. Δ_T was generated by Algorithm 1. Note both axes are log scales.

eters of the controller were (uniformly) randomly initialized in the range $[-0.1, 0.1]$ before each call to CONJPOMDP. After each call to CONJPOMDP, the average reward of the resulting controller was computed exactly by calculating the stationary distribution for the controller. From Figure 4, optimality is reliably achieved using approximately 100 iterations of the Markov chain.

4.1.4 Training directly on-line with OLPOMDP

The controller was also trained on-line using Algorithm 4 (OLPOMDP) with fixed step-sizes $\gamma_t = c$ with $c = 0.1, 1, 10, 100$. Reducing step-sizes of the form $\gamma_t = c/t$ were tried, but caused intolerably slow convergence. Figure 5 shows the performance of the controller (measured exactly as in the previous section) as a function of the total number of iterations of the Markov chain, for different values of the step-size c. The graphs are averages over 100 runs, with the controller's weights randomly initialized in the range $[-0.1, 0.1]$ at the start of each run. From the figure, convergence to optimal is about an order of magnitude slower than that achieved by CONJPOMDP, for the best step-size of $c = 1.0$. Step-sizes much greater that $c = 10.0$ failed to reliably converge to an optimal policy.

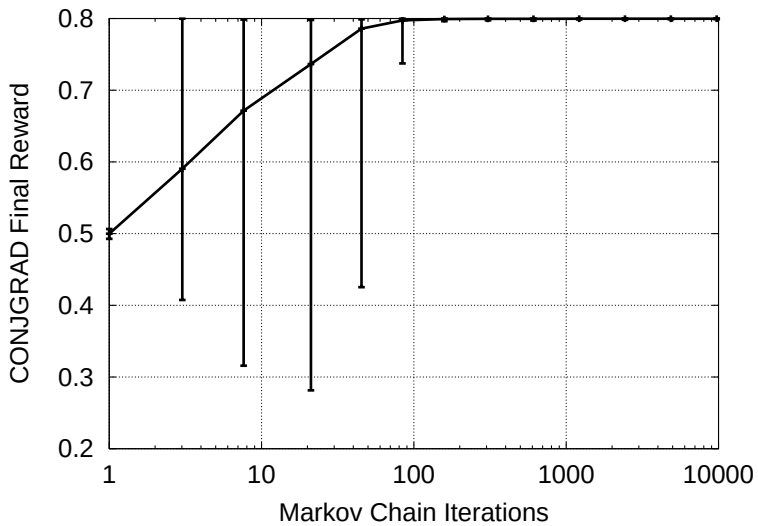

Figure 4: Performance of the 3-state Markov chain controller trained by CONJPOMDP as a function of the total number of iterations of the Markov chain. The performance was computed exactly from the stationary distribution induced by the controller. 0.8 is the average reward of the optimal policy. Averaged over 500 independent runs. The error bars were computed by dividing the results into two separate bins depending on whether they were above or below the mean, and then computing the standard deviation within each bin.

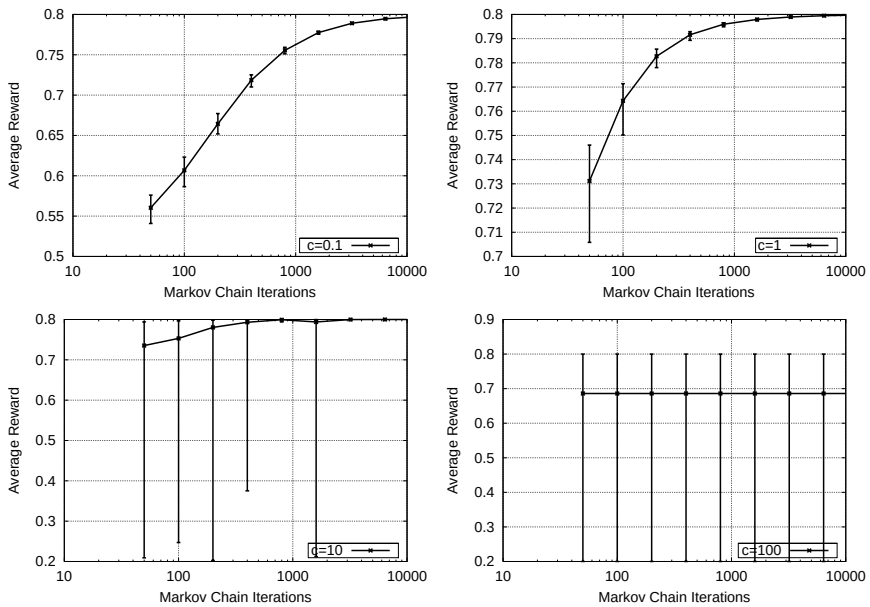

Figure 5: Performance of the 3-state Markov chain controller as a function of the number of iteration steps in the *on-line* algorithm, Algorithm 4, for fixed step sizes of $0.1, 1, 10$, and 100. Error bars were computed as in Figure 4.

4.2 Puck World

In this section, experiments are described in which $\mathrm{CONJPOMDP}$ and $\mathrm{OLPOMDP}$ were used to train 1-hidden-layer neural-network controllers to navigate a small puck around a two-dimensional world.

4.2.1 The World

The puck was a unit-radius, unit-mass section of a cylinder constrained to move in the plane in a region 100 units square. The puck had no internal dynamics (i.e rotation). Collisions with the region's boundaries were inelastic with a (tunable) coefficient of restitution e (set to 0.9 for the experiments reported here). The puck was controlled by applying a 5 unit force in either the positive or negative x direction, and a 5 unit force in either the positive or negative y direction, giving four different controls in total. The control could be changed every $1/10$ of a second, and the simulator operated at a granularity of $1/100$ of a second. The puck also had a retarding force due to air resistance of $0.005 \times \text{speed}^2$. There was no friction between the puck and the ground.

The puck was given a reward at each decision point ($1/10$ of a second) equal to $-d$ where d was the distance between the puck and some designated target point. To encourage the controller to learn to navigate the puck to the target independently of the starting state, the puck state was reset every 30 (simulated) seconds to a random

location and random x and y velocities in the range $[-10, 10]$, and at the same time the target position was set to a random location.

Note that the size of the state-space in this example is essentially infinite, being of the order of $2^{\text{PRECISION}}$ where PRECISION is the floating point precision of the machine (64 bits).

4.2.2 The controller

A one-hidden-layer neural-network with six input nodes, eight hidden nodes and four output nodes was used to generate a probabilistic policy in a similar manner to the controller in the three-state Markov chain example of the previous section. Four of the inputs were set to the raw x and y locations and velocities of the puck at the current time-step, the other two were the differences between the puck's x and y location and the target's x and y location respectively. The location inputs were scaled to lie between -1 and 1, while the velocity inputs were scaled so that a speed of 10 units per second mapped to a value of 1. The hidden nodes computed a $\tanh$ squashing function, while the output nodes were linear. Each hidden and output node had the usual additional offset parameter. The four output nodes were exponentiated and then normalized as in the Markov-chain example to produce a probability distribution over the four controls (± 5 units thrust in the x direction, ± 5 units thrust in the y direction). Controls were selected at random from this distribution.

4.2.3 Conjugate gradient ascent

We trained the neural-network controller using CONJPOMDP with the gradient estimates generated by GPOMDP. After some experimentation we chose $\beta = 0.95$ and $T = 1,000,000$ as the parameters CONJPOMDP supplied to GPOMDP. GSEARCH used the same value of β and the scheme discussed in Section 4.1.3 to determine the number of iterations with which to call GPOMDP.

Due to the saturating nature of the neural-network hidden nodes (and the exponentiated output nodes), there was a tendency for the network weights to converge to local minima at "infinity". That is, the weights would grow very rapidly early on in the simulation, but towards a suboptimal solution. Large weights tend to imply very small gradients and thus the network becomes "stuck" at these suboptimal solutions. We have observed a similar behaviour when training neural networks for pattern classification problems. To fix the problem, we subtracted a small quadratic penalty term $\gamma||\theta||^2$ from the performance estimates and hence also a small correction $2\gamma\theta_i$ from the gradient calculation[3] for θ_i.

We used a decreasing schedule for the quadratic penalty weight γ (arrived at through some experimentation). γ was initialized to 0.5 and then on every tenth iteration of CONJPOMDP, if the performance had improved by less than 10% from the value ten iterations ago, γ was reduced by a factor of 10. This schedule solved nearly all the local minima problems, but at the expense of slower convergence of the controller.

[3] When used as a technique for capacity control in pattern classification, this technique goes by the name "weight decay". Here we used it to condition the optimization problem.

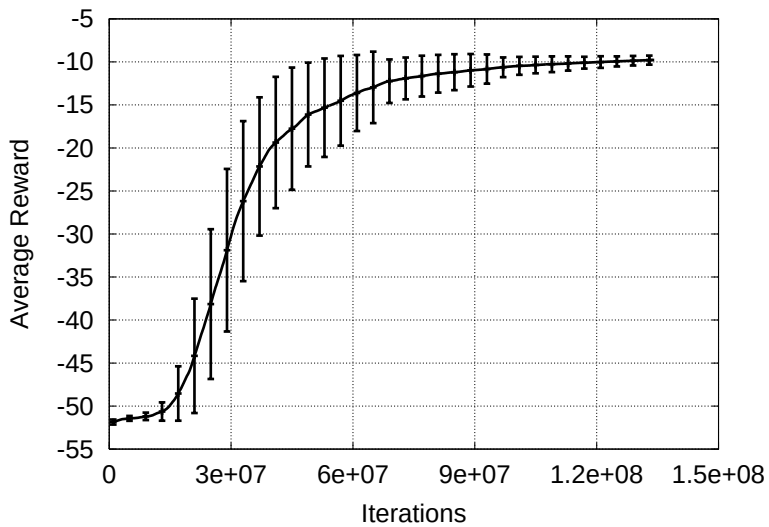

Figure 6: Performance of the neural-network puck controller as a function of the number of iterations of the puck world, when trained using CONJPOMDP. Performance estimates were generated by simulating for $1,000,000$ iterations. Averaged over 100 independent runs (excluding the four bad runs in Figure 7).

A plot of the average reward of the neural-network controller is shown in Figure 6, as a function of the number of iterations of the POMDP. The graph is an average over 100 independent runs, with the parameters initialized randomly in the range $[-0.1, 0.1]$ at the start of each run. The bad runs shown in Figure 7 were omitted from the average because they gave misleadingly large error bars.

Note that the optimal performance (within the neural-network controller class) seems to be around -8 for this problem, due to the fact that the puck and target locations are reset every 30 simulated seconds and hence there is a fixed fraction of the time that the puck must be away from the target. From Figure 6 we see the final performance of the puck controller is close to optimal. In only 4 of the 100 runs did CONJPOMDP get stuck in a suboptimal local minimum. Three of those cases were caused by overshooting in GSEARCH (see Figure 7), which could be prevented by adding extra checks to CONJPOMDP.

Figure 8 illustrates the behaviour of a typical trained controller. For the purpose of the illustration, only the target location and puck velocity were randomized every 30 seconds, not the puck location.

4.3 Call Admission Control

In this section we report the results of experiments in which CONJPOMDP was applied to the task of training a controller for the call admission problem treated in [8, Chapter

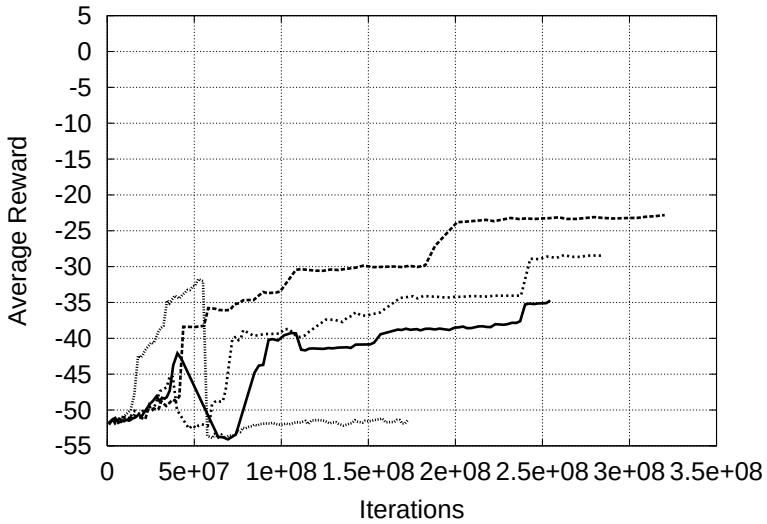

Figure 7: Plots of the performance of the neural-network puck controller for the four runs (out of 100) that converged to substantially suboptimal local minima.

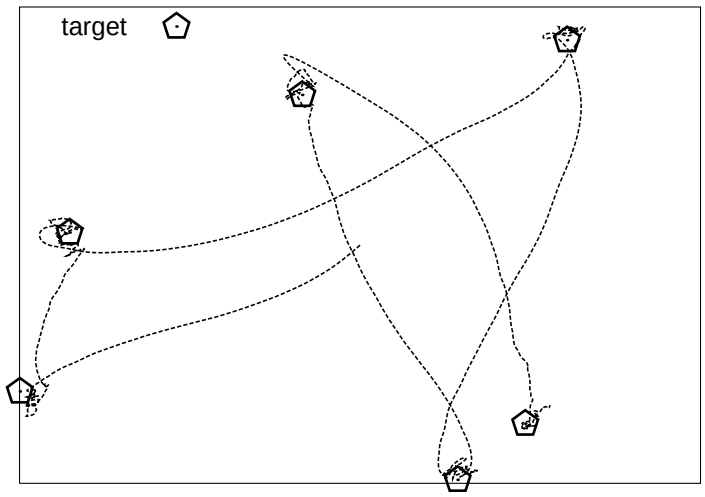

Figure 8: Illustration of the behaviour of a typical trained puck controller.

Call Type		1	2	3
Bandwidth Demand	b	1	1	1
Arrival Rate	α	1.8	1.6	1.4
Average Holding Time	h	0.6	0.5	0.4
Reward	r	1	2	4

Table 3: Parameters of the call admission control problem.

7].

4.3.1 The Problem

The call admission control problem treated in [8, Chapter 7] models the situation in which a telecommunications provider wishes to sell bandwidth on a communications link to customers in such a way as to maximize long-term average reward.

Specifically, the problem is a queuing problem. There are three different types of call, each with its own call arrival rate $\alpha(1)$, $\alpha(2)$, $\alpha(3)$, bandwidth demand $b(1)$, $b(2)$, $b(3)$ and average holding time $h(1)$, $h(2)$, $h(3)$. The arrivals are Poisson distributed while the holding times are exponentially distributed. The link has a maximum bandwidth of 10 units. When a call arrives and there is sufficient available bandwidth, the service provider can choose to accept or reject the call (if there is not enough available bandwidth the call is always rejected). Upon accepting a call of type m, the service provider receives a reward of $r(m)$ units. The goal of the service provider is to maximize the long-term average reward.

The parameters associated with each call type are listed in Table 3. With these settings, the optimal policy (found by dynamic programming in [8]) is to always accept calls of type 2 and 3 (assuming sufficient available bandwidth) and to accept calls of type 1 if the available bandwidth is at least 3. This policy has an average reward of 0.804, while the "always accept" policy has an average reward[4] of 0.784.

4.3.2 The Controller

As in [8], the controller had three parameters $\theta = (\theta_1, \theta_2, \theta_3)$, one for each type of call. Upon arrival of a call of type m, the controller chooses to accept the call with probability

$$\mu(\theta) = \begin{cases} \frac{1}{1+\exp(1.5(b-\theta_m))} & \text{if } b + b(m) \leq 10, \\ 0 & \text{otherwise,} \end{cases}$$

where b is the currently used bandwidth. This is the class of controllers studied in [8].

[4]There is some discrepancy between our average rewards and those quoted in [8]. This is probably due to a discrepancy in the way the state transitions are counted, which was not clear from the discussion in [8].

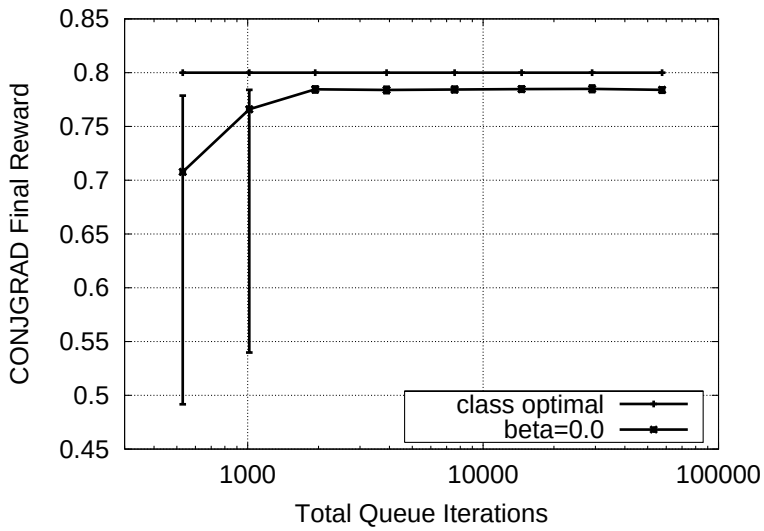

Figure 9: Performance of the call admission controller trained by CONJPOMDP as a function of the total number of iterations of the queue. The performance was computed by simulating the controller for 100,000 iterations. The average reward of the globally optimal policy is 0.804, the average reward of the optimal policy within the class is 0.8, and the plateau performance of CONJPOMDP is 0.784. The graphs are averages from 100 independent runs.

4.3.3 Conjugate gradient ascent

CONJPOMDP was used to train the above controller, with GPOMDP generating the gradient estimates from a range of values of β and T. The influence of β on the performance of the trained controllers was marginal, so we set $\beta = 0.0$ which gave the lowest-variance estimates. We used the same value of T for calls to GPOMDP within CONJPOMDP and within GSEARCH, and this was varied between 10 and $10,000$. The controller was always started from the same parameter setting $\theta = (8, 8, 8)$ (as was done in [8]). The value of this initial policy is 0.691. The graph of the average reward of the final controller produced by CONJPOMDP as a function of the total number of iterations of the queue is shown in Figure 9. A performance of 0.784 was reliably achieved with less than 2000 iterations of the queue.

Note that the optimal policy is not achievable with this controller class since it is incapable of implementing any threshold policy other than the "always accept" and "always reject" policies. Athough not provably optimal, a parameter setting of $\theta_1 \approx 7.5$ and any suitably large values of θ_2 and θ_3 (we chose $\theta_2 = \theta_3 = 15$) generates something close to the optimal policy within the controller class, with an average reward of 0.8. Figure 10 shows the probability of accepting a call of each type under this policy, as a function of the available bandwidth.

The controllers produced by CONJPOMDP with $\beta = 0.0$ and sufficiently large T

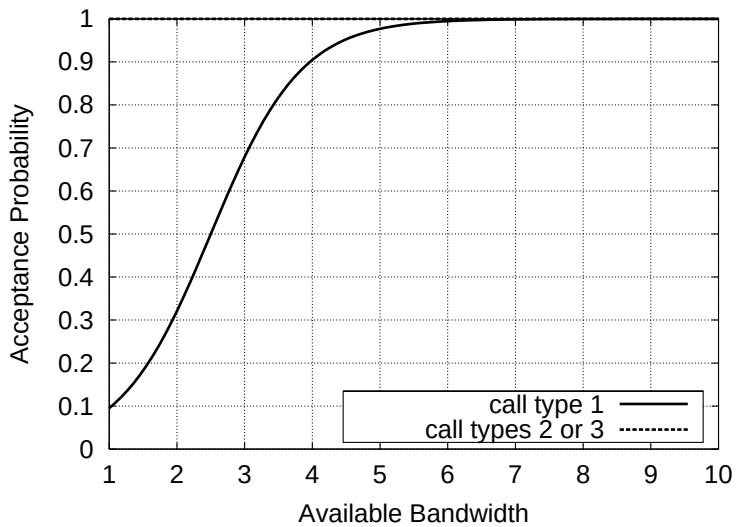

Figure 10: Probability of accepting a call of each type under the call admission policy with near-optimal parameters $\theta_1 = 7.5, \theta_2 = \theta_3 = 15$. Note that calls of type 2 and 3 are essentially always accepted.

are essentially "always accept" controllers with an average reward of 0.784, within 2% of the optimum achievable in the class. To produce policies even nearer to the optimal policy in performance, CONJPOMDP must keep θ_1 close to its starting value of 8, and hence the gradient estimate $\Delta_T = (\Delta_1, \Delta_2, \Delta_3)$ produced by GPOMDP must have a relatively small first component. Figure 11 shows a plot of normalized Δ_T as a function of β, for $T = 1,000,000$ (sufficiently large to ensure low variance in Δ_T) and the starting parameter setting $\theta = (8, 8, 8)$. From the figure, Δ_1 starts at a high value which explains why CONJPOMDP produces "always accept" controllers for $\beta = 0.0$, and does not become negative until $\beta \approx 0.93$, a value for which the variance in Δ_T even for moderately large T is relatively high.

A plot of the performance of CONJPOMDP for $\beta = 0.9$ and $\beta = 0.95$ is shown in Figure 12. Approximately half of the remaining 2% in performance can be obtained by setting $\beta = 0.9$, while for $\beta = 0.95$ a sufficiently large choice for T gives most of the remaining performance. For this problem, there is a huge difference between gaining 98% of optimal performance, which is achieved for $\beta = 0.0$ and less than 2000 iterations of the queue, and gaining 99% of the optimal which requires $\beta = 0.9$ and of the order of 500,000 queue iterations. A similar convergence rate and final approximation error to the latter case were reported for the on-line algorithms in [8, Chapter 7], although the results of only one run were given in each case.

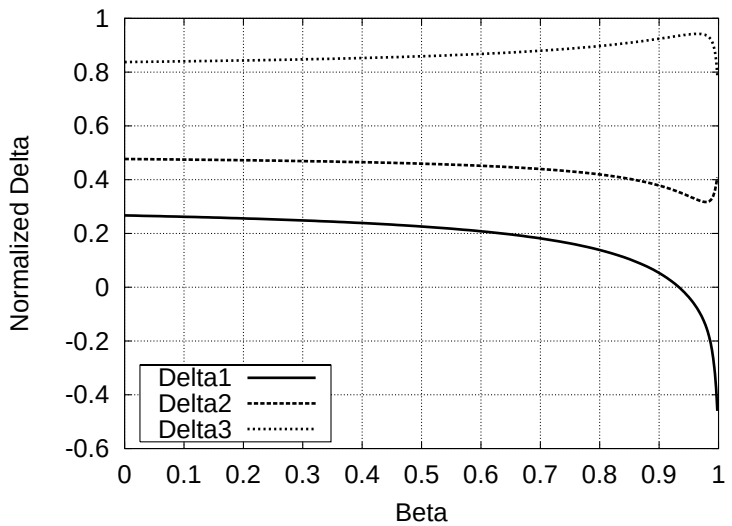

Figure 11: Plot of the three components of Δ_T for the call admission problem, as a function of the discount parameter β. The parameters were set at $\theta = (8, 8, 8)$. T was set to $1,000,000$. Note that Δ_1 does not become negative (the correct sign) until $\beta \approx 0.93$.

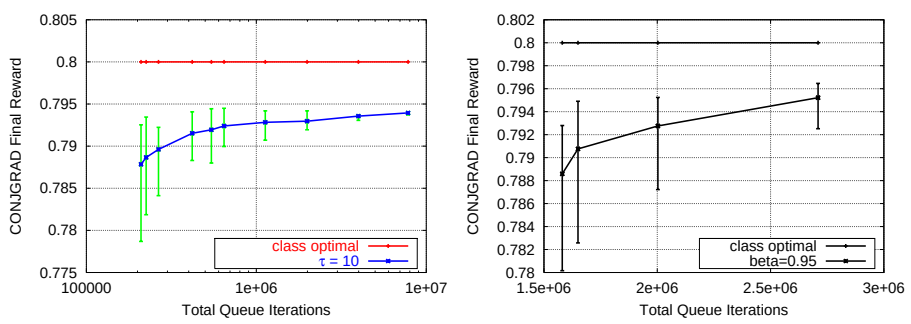

Figure 12: Performance of the call admission controller trained by CONJPOMDP as a function of the total number of iterations of the queue. The performance was calculated by simulating the controller for 1,000,000 iterations. The graphs are averages from 100 independent runs.

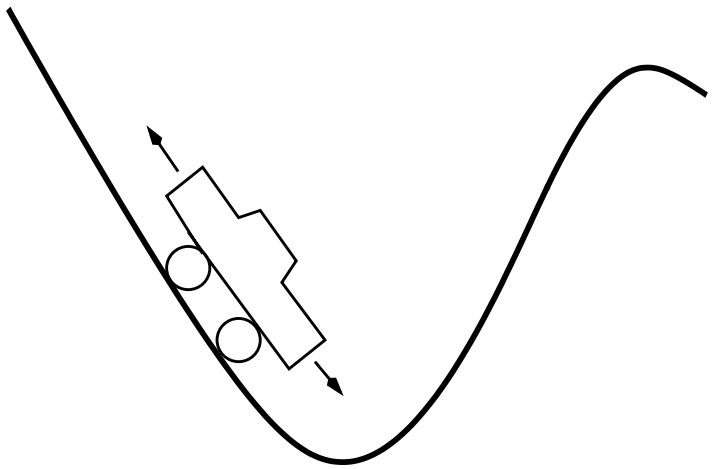

Figure 13: The classical "mountain-car" task is to apply forward or reverse thrust to the car to get it over the crest of the hill. The car starts at the bottom and does not have enough power to drive directly up the hill.

4.4 Mountainous Puck World

The "mountain-car" task is a well-studied problem in the reinforcement learning literature [13, Example 8.2]. As shown in Figure 13, the task is to drive a car to the top of a one-dimensional hill. The car is not powerful enough to accelerate directly up the hill against gravity, so any successful controller must learn to "oscillate" back and forth until it builds up enough speed to crest the hill.

In this section we describe a variant of the mountain car problem based on the puck-world example of Section 4.2. With reference to Figure 14, in our problem the task is to navigate a puck out of a valley and onto a plateau at the northern end of the valley. As in the mountain-car task, the puck does not have sufficient power to accelerate directly up the hill, and so has to learn to oscillate in order to climb out of the valley. Once again we were able to reliably train near-optimal neural-network controllers for this problem, using CONJPOMDP and GSEARCH, and with GPOMDP generating the gradient estimates.

4.4.1 The World

The world dimensions, physics, puck dynamics and controls were identical to the flat puck world described in Section 4.2, except that the puck was subject to a constant gravitational force of 10 units, the maximum allowed thrust was 3 units (instead of 5), and the height of the world varied as follows:

$$\text{height}(x, y) = \begin{cases} 15 & \text{if } y < 25 \text{ or } y > 75 \\ 7.5\left[1 - cos\left(\frac{\pi\left(\frac{y}{2}-50\right)}{25}\right)\right] & \text{otherwise.} \end{cases}$$

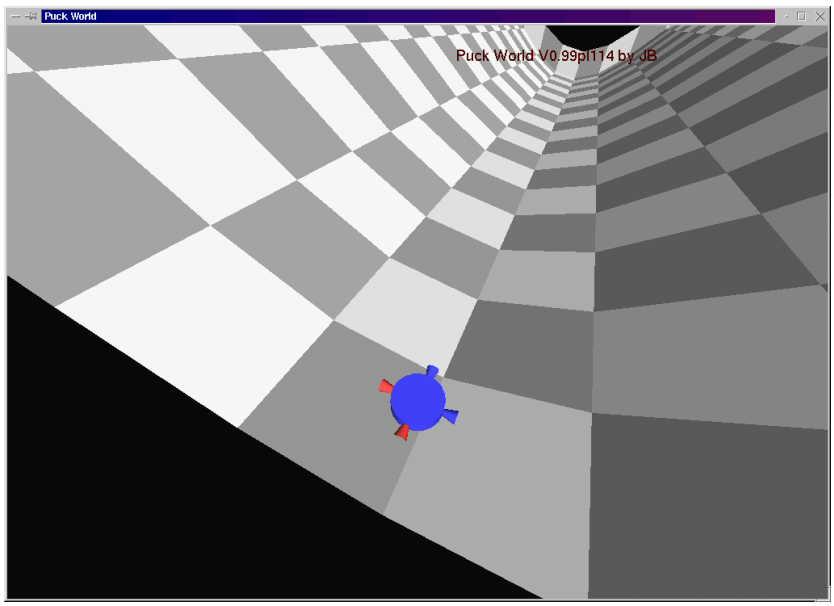

Figure 14: In our variant of the mountain-car problem the task is to navigate a puck out of a valley and onto the northern plateau. The puck starts at the bottom of the valley and does not have enough power to drive directly up the hill.

With only 3 units of thrust, a unit mass puck can not accelerate directly out of the valley.

Every 120 (simulated) seconds, the puck was initialized with zero velocity at the bottom of the valley, with a random x location. The puck was given no reward while in the valley or on the southern plateau, and a reward of $100 - s^2$ while on the northern plateau, where s was the speed of the puck. We found the speed penalty helped to improve the rate of convergence of the neural network controller.

4.4.2 The controller

After some experimentation we found that a neural-network controller could be reliably trained to navigate to the northern plateau, or to stay on the northern plateau once there, but it was difficult to combine both in the same controller (this is not so surprising since the two tasks are quite distinct). To overcome this problem, we trained a "switched" neural-network controller: the puck used one controller when in the valley and on the southern plateau, and then switched to a second neural-network controller while on the northern plateau. Both controllers were one-hidden-layer neural-networks with nine input nodes, five hidden nodes and four output nodes. The nine inputs were the normalized ($[-1, 1]$-valued) x, y and z puck locations, the normalized x, y and z locations relative to center of the northern wall, and the x, y and z puck velocities. The four outputs were used to generate a policy in the same fashion as the controller of

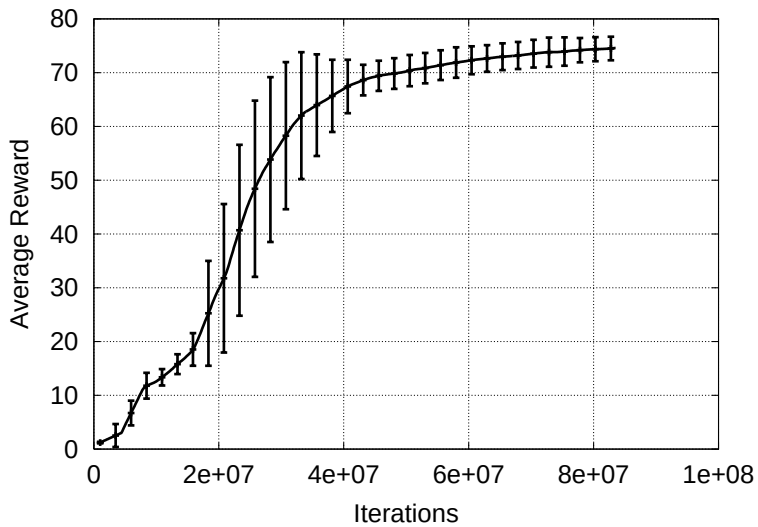

Figure 15: Performance of the neural-network puck controller as a function of the number of iterations of the mountainous puck world, when trained using CONJPOMDP. Performance estimates were generated by simulating for $1,000,000$ iterations. Averaged over 100 independent runs.

Section 4.2.2.

4.4.3 Conjugate gradient ascent

The switched neural-network controller was trained using the same scheme discussed in Section 4.2.3, except this time the discount factor β was set to 0.98.

A plot of the average reward of the neural-network controller is shown in Figure 15, as a function of the number of iterations of the POMDP. The graph is an average over 100 independent runs, with the neural-network controller parameters initialized randomly in the range $[-0.1, 0.1]$ at the start of each run. In this case no run failed to converge to near-optimal performance. From the figure we can see that the puck's performance is nearly optimal after about 40 million total iterations of the puck world. Although this figure may seem rather high, to put it in some perspective note that a random neural-network controller takes about 10,000 iterations to reach the northern plateau from a standing start at the base of the valley. Thus, 40 million iterations is equivalent to only about 4,000 trips to the top for a random controller.

Note that the puck converges to a final average performance around 75, which indicates it is spending at least 75% of its time on the northern plateau. Observation of the puck's final behaviour shows it behaves nearly optimally in terms of oscillating back and forth to get out of the valley.

5 Conclusion

This paper showed how to use the performance gradient estimates generated by the GPOMDP algorithm from [2] to optimize the average reward of parameterized POMDPs. The optimization relies on the use of GSEARCH, a robust line-search algorithm that uses gradient estimates, rather than value estimates to bracket the maximum. CONJPOMDP and GSEARCH were found to perform well on four quite distinct problems: optimizing a controller for a three-state MDP, optimizing a neural-network controller for navigating a puck around a two-dimensional world, optimizing a controller for a call admission problem, and optimizing a switched neural-network controller in a variation of the classical mountain-car task. We also presented OLPOMDP, an on-line version of CONJPOMDP.

For the three-state MDP and the call admission problems we were able to provide graphic illustrations of how the bias and variance of the gradient estimates $\nabla_\beta \eta$ can be traded against one another by varying β between 0 (low variance, high bias) and 1 (high variance, low bias).

Relatively little tuning was required to generate these results. In addition, the controllers operated on direct and simple representations of the state, in contrast to the much more complex representations usually required of value-function based approaches.

An interesting avenue for further research would be an empirical comparison of value-function based methods and the algorithms of this paper in domains where the former are known to produce good results.

Despite the success of CONJPOMDP/GSEARCH in the experiments described here, the on-line algorithm OLPOMDP has advantages in other settings. In particular, when it is applied to multi-agent reinforcement learning, both gradient computations and parameter updates can be performed for distinct agents without any communication beyond the global distribution of the reward signal. This idea has led to a biologically plausible parameter optimization procedure for spiking neural networks (see [1]), and we are currently investigating the application of the on-line algorithm in multi-agent reinforcement learning problems.

References

[1] P. L. Bartlett and J. Baxter. in preparation. September 1999.

[2] J. Baxter and P. L. Bartlett. Direct Gradient-Based Reinforcement Learning: I. Gradient Estimation Algorithms. Technical report, Research School of Information Sciences and Engineering, Australian National University, July 1999.

[3] J. Baxter, A. Tridgell, and L. Weaver. Learning to Play Chess Using Temporal-Differences. *Machine Learning*, 1999. To appear.

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

[5] X.-R. Cao and Y.-W. Wan. Algorithms for Sensitivity Analysis of Markov Chains Through Potentials and Perturbation Realization. *IEEE Transactions on Control Systems Technology*, 6:482–492, 1998.

[6] T. L. Fine. *Feedforward Neural Network Methodology*. Springer, New York, 1999.

[7] H. Kimura, K. Miyazaki, and S. Kobayashi. Reinforcement learning in POMDPs with function approximation. In D. H. Fisher, editor, *Proceedings of the Fourteenth International Conference on Machine Learning (ICML'97)*, pages 152–160, 1997.

[8] P. Marbach. *Simulation-Based Methods for Markov Decision Processes*. PhD thesis, Labortory for Information and Decision Systems, MIT, 1998.

[9] P. Marbach and J. N. Tsitsiklis. Simulation-Based Optimization of Markov Reward Processes. Technical report, MIT, 1998.

[10] A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.

[11] S. Singh and D. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, pages 974–980. MIT Press, 1997.

[12] R. Sutton. Learning to Predict by the Method of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[13] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge MA, 1998. ISBN 0-262-19398-1.

[14] G. Tesauro. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8:257–278, 1992.

[15] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[16] L. Weaver and J. Baxter. Reinforcement Learning From State and Temporal Differences. Technical report, Department of Computer Science, Australian National University, May 1999. http://csl.anu.edu.au/∼jon/papers/std_full.ps.gz.

[17] R. J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8:229–256, 1992.

[18] W. Zhang and T. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1114–1120. Morgan Kaufmann, 1995.