Solving Deep Memory POMDPs with Recurrent Policy Gradients

Daan Wierstra[1], Alexander Foerster[1], Jan Peters[2], Juergen Schmidhuber[1]

(1) IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland
(2) University of Southern California, Los Angeles, CA, USA

Abstract. This paper presents Recurrent Policy Gradients, a model-free reinforcement learning (RL) method creating limited-memory stochastic policies for partially observable Markov decision problems (POMDPs) that require long-term memories of past observations. The approach involves approximating a policy gradient for a Recurrent Neural Network (RNN) by backpropagating return-weighted characteristic eligibilities through time. Using a "Long Short-Term Memory" architecture, we are able to outperform other RL methods on two important benchmark tasks. Furthermore, we show promising results on a complex car driving simulation task.

1 Introduction

Policy gradient (PG) methods have been among the few successful algorithms for solving real world Reinforcement Learning (RL) tasks (e.g. see [1–5]) as they can deal with continuous states and actions. After being introduced by Williams [6] and Gullapalli [7], they have recently been extended to deal more efficiently [8] with complex high-dimensional tasks [9]. Although sensitive to local minima in policy representation space, an attractive property of this class of algorithms is that they are guaranteed to converge to at least a locally optimal policy, even using function approximators such as neural networks [10]. Furthermore, unlike most regular RL approaches, they provide a natural framework for learning policies with *continuous*, high-dimensional actions. Provided the choice of policy representation is powerful enough, PGs can tackle arbitrary RL problems. Unfortunately, most PG approaches have only been used to train policy representations of, typically, a few dozen parameters at most. Surprisingly, the obvious combination with standard backpropagation techniques has not been extensively investigated (a notable exception being the SRV algorithm [7, 11]). In this paper, we address this shortcoming, and show how PGs can be naturally combined with backpropagation, and BackPropagation Through Time (BPTT) [12] in particular, to form a powerful RL algorithm capable of training complex neural networks with large numbers of parameters.

RL tasks in realistic environments typically need to deal with incomplete and noisy state information resulting from partial observability such as encountered in POMDPs. Furthermore, they often need to deal with non-Markovian

problems where there are dependencies onto significantly earlier states. Both POMDPs and non-Markovian problems largely defy traditional value function based approaches and require complex state estimators with internal memory based on accurate knowledge of the system. In reinforcement learning, we can rarely assume a good memory-based state estimator as the underlying system is usually unknown.

A naive alternative to using memory is learning *reactive stochastic policies* [13] which simply map observations to probabilities of actions. The underlying assumption is that state-information does not play a crucial role during most parts of the problem and that using random actions can prevent the actor from getting stuck in an endless loop for ambiguous observations (e.g., the scenario of a blind robot rolling forever against the wall would only happen for a stationary, deterministic reactive policy). In general, this strategy is clearly suboptimal, and instead algorithms that use some form of memory still seem essential.

However, if we cannot assume a perfect model and a precisely estimated state, an optimal policy needs to be both stochastic and memory-based for realistic environments. As the memory is always an abstraction or imperfect, i.e., limited, we will focus on learning what we define as *limited-memory stochastic policies*, that is, policies that map limited memory states to probability distributions on actions. Work on policy gradient methods with memory has been scarce so far, largely limited to finite state controllers [14, 15]. In this paper, we extend this approach to more sophisticated policy representations capable of representing memory using an RNN architecture called Long Short-Term Memory (LSTM), see [16]. We develop a new reinforcement learning algorithm that can effectively learn memory-based policies for deep memory POMDPs. We present *Recurrent* Policy Gradients (RPG), an algorithm which backpropagates the estimated return-weighted *eligibilities* backwards in time through recurrent connections in the RNN. As a result, policy updates can become a function of any event in the history. We show that the presented method outperforms other RL methods on two important RL benchmark tasks with different properties: continuous control in a non-Markovian double pole balancing environment, and discrete control on the deep memory T-maze [17] task. Moreover, we show promising results in a complex car driving simulation.

The structure of the paper is as follows. The next section describes the Recurrent Policy Gradient algorithm. The subsequent sections describe our experimental results using RPGs with memory and conclude with a discussion.

2 Recurrent Policy Gradients

In this section we describe our basic algorithm for using memory in a policy gradient setting. First, we briefly summarize reinforcement learning terminology, then we briefly review the policy gradient framework. Next, we describe the particular type of recurrent neural network architecture used in this paper, Long

Short-Term Memory. Subsequently, we show how the methods can be combined to form Recurrent Policy Gradients.

2.1 Reinforcement Learning – Generalized Problem Statement

First let us introduce the RL framework used in this paper and the corresponding notation. The environment produces a state g_t at every time step. Transitions from state to state are governed by a probability function $p(g_{t+1}|a_{1:t}, g_{1:t})$ unknown to the agent but dependent upon all previous actions $a_{1:t}$ executed by the agent and all previous states $g_{1:t}$ of the system. Note that most reinforcement learning papers need to assume Markovian environments – we will later see that we do not need to for policy gradient methods with an internal memory. Let r_t be the reward assigned to the agent at time t, and let o_t be the corresponding observation produced by the environment. We assume that both quantities are governed by fixed distributions $p(o|g)$ and $p(r|g)$, solely dependent on state g.

In the more general reinforcement setting, we require that the agent has a memory of the generated experience consisting of finite episodes. Such episodes are generated by the agent's operations on the (stochastic) environment, executing action a_t at every time step t, after observing observation o_t and special 'observation' r_t (the reward) which both depend solely on g_t. We define the *observed history*[1] h_t as the string or vector of observations and actions up to moment t since the beginning of the episode: $h_t = \langle o_0, a_0, o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t \rangle$. The complete history H includes the unobserved states and is given by $H_T = \langle h_T, g_{0:T} \rangle$. At any time t, the actor optimizes $R_t = (1-\gamma)^{-1} \sum_{k=t}^{\infty} r_k \gamma^{t-k-1}$ which is the *return* at at time t where $0 < \gamma < 1$ denotes a discount factor.

The expectation of this return R_t at time $t = 0$ is also the measure of quality of our policy and, thus, the objective of reinforcement learning is to determine a policy which is optimal with respect to the expected future discounted rewards or expected return $J = E[R_0] = (1-\gamma)^{-1} \lim_{T\to\infty} \mathbf{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$. For the average reward case where $\gamma \to 1$ or, equivalently, $(1-\gamma)^{-1} \to \infty$, this expression remains true analytically but needs to be replaced by $J = \lim_{T\to\infty} \mathbf{E}[\sum_{t=0}^{T-1} r_t/T]$ in order to be numerically feasible.

An optimal or near-optimal policy in a non-Markovian or partially observable Markovian environment requires that the action a_t is taken depending on the *entire* preceding history. However, in most cases, we will not *need* to store the whole string of events but only sufficient statistics $T(h_t)$ of the events which we call the limited memory of the agents past. Thus, a stochastic policy π can be defined as $\pi(a|h_t) = p(a|T(h_t); \theta)$, implemented as an RNN with weights θ and stochastically interpretable output neurons. This produces a probability distribution over actions, from which actions a_t are drawn $a_t \sim \pi(a|h_t)$.

[1] Note that such histories are also called path or trajectory in the literature.

2.2 The Policy Gradient Framework

The type of RL algorithm we employ in this paper falls in the class of policy gradient algorithms, which, unlike many other (notably TD) methods, update the agent's policy-defining parameters θ directly by estimating a gradient in the direction of higher (average or discounted) reward.

Now, let $R(H)$ be some measure of the total reward accrued during a history (e.g., $R(H)$ could be the average of the rewards for the average reward case or the discounted sum for the discounted case), and let $p(H|\theta)$ be the probability of a history given policy-defining weights θ, then the quantity the algorithm should be optimizing is $J = \int_H p(H|\theta)R(H)dH$. This, in essence, indicates the expected reward over all possible histories, weighted by their probabilities under π. Using gradient ascent to update parameters θ of policy π, we can write

$$\nabla_\theta J = \int \nabla_\theta p(H) R(H) dH = \int \frac{p(H)}{p(H)} \nabla_\theta p(H) R(H) dH = \int p(H) \nabla_\theta \log p(H) R(H) dH$$

using the "likelihood-ratio trick" and the fact that $\nabla_\theta R(H) = 0$ for a single, fixed H. Taking the sample average as Monte Carlo (MC) approximation of this expectation by taking N trial histories we get

$$\nabla_\theta J = \mathbf{E}_H \Big[\nabla_\theta \log p(H) R(H) \Big] \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \log p(H^n) R(H^n).$$

which is a fast approximation of the policy gradient for the current policy with the convergence speed of $O(N^{-1/2})$ to the true gradient independent of the number of parameters of the policy (i.e., number of elements of the gradient).

Probabilities of histories $p(H)$ are dependent on an unknown initial state distribution, on unknown observation probabilities per state, and on unknown state transition function $p(g_{t+1}|a_{1:t}, g_{1:t})$. But at least the agent knows its own action probabilities, so the log derivative for agent parameters θ in $\nabla_\theta \log p(h)$ can be acquired by first realizing that the probability of a particular history is the product of all actions and observations given subhistories:

$$p(H_T) = p(\langle o_0, g_0 \rangle) \prod_{t=1}^{T} p(\langle o_t, g_t \rangle | h_{t-1}, a_{t-1}, g_{0:t}) \pi(a_{t-1}|h_{t-1})$$

Taking the log-derivative results into transforming this large product into a sum $\log p(H_T) = (\text{const}) + \sum_{t=0}^{T} \log \pi(a_t|h_t)$: where most parts are not affected by θ, i.e., are constant. Thus, when taking the derivative of this term, we obtain

$$\nabla_\theta \log p(H_T) = \sum_{t=0}^{T} \nabla_\theta \log \pi(a_t|h_t).$$

Substituting this term into our MC approximation results in a gradient estimator which only requires observed variables. However, if we make use of the fact that

future actions do not depend on past rewards, we can show that these terms can be omitted from the gradient estimate (see [5] for details). Thus, an unbiased gradient estimator is given by

$$\nabla_\theta J \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi(a_t|h_t^n) R_t^n$$

which yields the desired gradient estimator which only has observable variables.

Nevertheless, an important problem with this Monte Carlo approach is the often high variance in the gradient estimate. For example, if $R(h) = 1$ for all h, the variance can be given by $\sigma_T^2 = E[\sum_{t=0}^{T}(\nabla_\theta \log \pi(a_t|h_t^n))^2]$ which grows linearly with T. One way to tackle such problems and reduce this variance is to include a *baseline* b (first introduced by Williams [6]) into the gradient estimate $\nabla_\theta J \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \log p(h_t^n)(R_t^n - b)$ where b is typically the expected average reward or some kind of value function. Due to the likelihood-ratio trick $\int p(H) \nabla_\theta \log p(H) R(H) dH = \nabla_\theta \int p(H) b dH = \nabla_\theta 1 = 0$, we can guarantee that $E[\sum_{n=1}^{N} \nabla_\theta \log p(H_t^n) b] = 0$ and, thus, the baseline can only reduce the variance but not bias the gradient in any way [6].

Another way to reduce the variance of the gradient is to reduce the planning horizon of the agent, i.e., the episode length can be truncated or we can use a discount factor $\hat{\gamma}$ which is lower than the true γ as in the GPOMDP algorithm [18] where $0 < \hat{\gamma} < \gamma$. Unfortunately, here can be a big trade-off as $\hat{\gamma} < \gamma$ biases the gradient, but when $\hat{\gamma} \to \gamma$, this bias vanishes while the gradient estimate variance can increase significantly.

2.3 LSTM Recurrent Function Approximators

RNNs have attracted some attention in the past decade because of their simplicity and potential power. However, though powerful in theory, they turn out to be quite limited in practice due to their inability to capture long-term time dependencies – they suffer from the problem of *vanishing gradient* [19], the fact that the gradient signal vanishes as the error signal is propagated back through time. Because of this, events more than 10 time steps apart can typically not be related.

One method purposely designed to avoid this problem is Long Short-Term Memory (LSTM [16]), which constitutes a special RNN architecture capable of capturing long term time dependencies. The defining feature of this architecture is that it consists of a number of *memory cells*, which can be used to store activations arbitrarily long. Access to the memory cell is *gated* by units that learn to open or close depending on the context.

LSTM networks have been shown to outperform other RNNs on numerous time series requiring the use of deep memory [20]. Therefore, they seem well-suited for usage in PG algorithms for complex, deep memory requiring tasks. Whereas RNNs are usually used to *predict*, we use them to *control* an agent directly, to represent a controller's policy receiving observations and producing action probabilities at every time step.

2.4 Introducing the Recurrency in Recurrent Policy Gradients

Typically, PG algorithms learn to map observations to action probabilities, i.e. they learn stochastic reactive policies. As noted before, this is clearly suboptimal for all but the simplest partial observability problems. We would like to equip our algorithm with adaptable memory, using LSTM to map histories or *memory states* to action probabilities. Unlike earlier methods, our method makes full use of the backpropagation technique while doing this: whereas most if not all published and experimentally tested PG methods (as far as the authors are aware) estimate parameters θ individually, we use *eligibility-backpropagation through time* (as opposed to standard error-backpropagation or BPTT [12]) to update all parameters conjunctively, yielding solutions that better generalize over complex histories. Using this method, we can map *histories* to actions instead of *observations* to actions.

In order to estimate the gradient for a history-based approach, we map histories h_t to action probabilities by using LSTM's internal state representation. Backpropagating return-weighted eligibilities [6] affects the policy such that it makes histories that were better than other histories (in terms of reward) more likely by reinforcing the probabilities of taking similar actions for similar histories.

Recurrent Policy Gradients are architecturally equal to supervised RNNs, however, the output neurons are interpreted as a probability distribution. It takes, at every time step during the forward pass of BPTT, as input observation o_t and reward r_t. Together with the recurrent connections, these produce outputs $\pi(h_t)$, representing the probability distribution on actions.

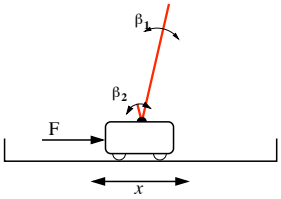
Only the output part of the neural network is interpreted stochastically. This allows us, during the backward pass, to only estimate the eligibilities of the output units at every time step. The gradient on the other parameters θ can be derived efficiently via eligibility backpropagation through time, treating output eligibilities like we would treat normal errors ('deltas') in an RNN trained with gradient descent. Also, by having only stochastic output units, we do not have to compute complicated gradients on stochastic internal (belief) states such as is done in [14, 15] – eligibility backpropagation through time disambiguates relevant hidden state automatically, when possible.

3 Experiments

We carried out experiments on three fundamentally different problem domains. The first task, double pole balancing with incomplete state information, is a *continuous* control task that has been a benchmark in the RL community for many years. The second task, the T-maze, is a difficult discrete control task that requires remembering its initial observation until the end of the episode. The third task involves a complex car racing simulator, called Torcs.

All experiments were carried out with 10-cell LSTMs. The baseline estimator used was simply a moving average of the return received at any time step.

3.1 Continuous Control: (PO)MDP (Double) Pole Balancing



	Markov	non-Markov
1 pole	863 ± 213	1893 ± 527
2 poles	4981 ± 1386	5649 ± 1548

Fig. 1. The non-Markov double pole balancing task. The results show the mean and standard deviation of the number of evaluations until the success criterion.

This task involves trying to balance a pole hinged on a cart that moves on a finite track (see Figure 1). The single control consists of the force F applied to the cart (in Newtons), and observations usually include the cart's position x, the pole's angle θ and velocities $\dot{x}$ and $\dot{\theta}$. It provides a perfect testbed for algorithms focussing on learning fine control in continuous state and action spaces. However, recent successes in the RL field have made the standard pole balancing setup too easy and therefore obsolete. To make the task more challenging, we (1) remove velocity information $\dot{x}$ and $\dot{\theta}$ such that the problem becomes non-Markov, and (2) add a second pole to the same cart, of length 1/10th of the original one. This yields non-Markovian double pole balancing [21], a truly challenging task that has not been solved by any other single agent RL method but RPGs.

We applied RPGs to the pole balancing task, using a Gaussian output structure, consisting of a mean μ (which was interpreted linearly) and a standard deviation σ (which was scaled with the logistic function in order to prevent variances from being negative) where eligibilities were calculated according to [6]. We use a learning rate $\alpha\sigma^2$ (as suggested by Williams [6]) and $\alpha = 0.001$, $momentum = 0.9$ and $\gamma = 0.99$. Initial parameters θ were initialized randomly between -0.01 and 0.01. Reward was always 0.0, except for the last time step when one of the poles falls over, where it is -1.0.

A run was considered a *success* when the pole(s) did not fall over for $10,000$ time steps. Figure 1 shows results averaged over 20 runs. RPGs clearly outperform earlier PG methods, even by orders of magnitude (compare [15]'s finite state controller).

3.2 Discrete Control: the Long Term Dependency T-maze

The second experiment was carried out on the T-maze [17] (see Figure 2). Designed to test an RL algorithm's ability to correlate events far apart in history, it involves having to *learn* to remember the observation from the first time step until the episode ends. At the first time step, it starts at position **S** and perceives the **X** either north or south – meaning that the goal state **G** is in the north or

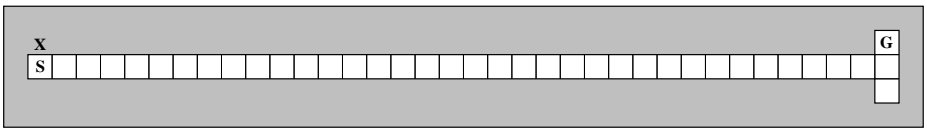


Fig. 2. The T-maze task. In this example, N, the length of the alley, is 35.

south part of the T-junction, respectively. Additionally, the agent perceives its immediate surroundings. The agent has four possible actions: North, East, South and West. These discrete actions are represented in the network as a softmax layer. When the agent makes the correct decision at the T-junction, i.e. go south if the **X** was south and north otherwise, it receives a reward of 4.0, otherwise a reward of -0.1. In both cases, this ends the episode. Note that the corridor length N can be increased to make the problem more difficult, since the agent has to learn to remember the initial 'road sign' for $N+1$ time steps. In Figure 2 we see an example T-maze with corridor length 35.

Corridor length N was systematically varied from 10 to 100, and for each length 10 runs were performed. Training was performed in batches of 20 normalizing the gradient to length 0.3. Discount factor $\gamma = 0.98$ was used. In Figure 3 the results are displayed, in addition to other algorithms' results taken from [17]. We can see that RPGs clearly outperform the value-based methods. This might be due to the difference in complexity of a simple policy versus an unnecessarily complex value function.

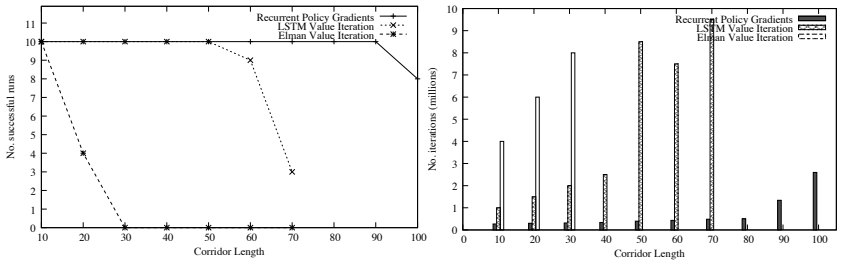


Fig. 3. T-maze results. The left chart shows the number of successful runs for $N = 10, \ldots, 100$. Recurrent Policy Gradients' performance starts to degrade at length $N = 100$. The right plot shows the number of average iterations required to solve the task, averaged over the successful runs. RPGs clearly outperform other RL methods on this task, to the best of the authors' knowledge. (The results for the Value Iteration based algorithms are taken from [17]).

4 The TORCS Car Racing Simulator

Fig. 4. The TORCS racing car simulator.

In order to test our algorithm's performance on a more complicated environment, we carried out experiments on the TORCS [22] car racing simulator. Observations are speed, steering angle, position and look-ahead-distance. The agent has to learn to drive and stay on the road while maintaining high speed. Its reward consists of the speed, but it gets punished for getting off the track. Maximum speed starts off with 10 km/h, and is gradually increased.

Our current experiments, on 5 trials, show the agent can consistently learn to steer and stay on the road after just under 2 minutes of real-time behavior. The agent can learn to drive safely – not getting off track – up to 70 km/h, after which its behavior destabilizes. The fastest lap time is just under 3 minutes, which is still twice as slow as a trained human player or our preprogrammed agent.

5 Discussion

We introduced a new policy gradient method equipped with memory capable of memorizing events from arbitrarily far in the past. The approach involves computing and backpropagating a policy gradient through time with LSTM representing memory, thus updating a policy which maps event histories to action probabilities. The approach outperformed other RL methods on two important benchmarks. We think Recurrent Policy Gradients constitute one of the most efficient RL algorithms to date on difficult non-Markovian tasks.

Acknowledgments

This research was funded by SNF grant 200021-111968/1 and by the 6th FP of the EU (project number IST-511931).

References

1. Benbrahim, H., Franklin, J.: Biped dynamic walking using reinforcement learning. Robotics and Autonomous Systems Journal (1997)
2. Moody, J., Saffell, M.: Learning to Trade via Direct Reinforcement. IEEE Transactions on Neural Networks **12**(4) (2001) 875–889
3. Prokhorov, D.: Toward effective combination of off-line and on-line training in adp framework. In: Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL). (2007)
4. Baxter, J., Bartlett, P., Weaver, L.: Experiments with infinite-horizon, policy-gradient estimation. Journal of Artificial Intelligence Research **15** (2001) 351–381
5. Peters, J., Schaal, S.: Policy gradient methods for robotics. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Beijing, China (2006) 2219 – 2225
6. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning **8** (1992) 229–256
7. Gullapalli, V.: A stochastic reinforcement learning algorithm for learning real-valued functions. Neural Networks **3**(6) (1990) 671–692
8. Schraudolph, N., Yu, J., Aberdeen, D.: Fast online policy gradient learning with smd gain vector adaptation. In Weiss, Y., Schölkopf, B., Platt, J., eds.: Advances in Neural Information Processing Systems 18. MIT Press, Cambridge, MA (2006)
9. Peters, J., Vijayakumar, S., Schaal, S.: Natural actor-critic. In: Proceedings of the 16th European Conference on Machine Learning (ECML 2005). (2005) 280–291
10. Sutton, R., McAllester, D., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation (2001)
11. Gullapalli, V.: Reinforcement learning and its application to control (1992)
12. Werbos, P.: Back propagation through time: What it does and how to do it. In: Proceedings of the IEEE. Volume 78. (1990) 1550–1560
13. Singh, S.P., Jaakkola, T., Jordan, M.I.: Learning without state-estimation in partially observable markovian decision processes. In: International Conference on Machine Learning. (1994) 284–292
14. Aberdeen, D.: Policy-Gradient Algorithms for Partially Observable Markov Decision Processes. PhD thesis, Australian National University (2003)
15. Meuleau, N., Peshkin, L., Kim, K.E., Kaelbling, L.P.: Learning finite-state controllers for partially observable environments. In: Proc. Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI '99). Morgan Kaufmann. (1999) 427–436
16. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8) (1997) 1735–1780
17. Bakker, B.: Reinforcement learning with long short-term memory. In: Advances in Neural Information Processing Syst., 14. (2002)
18. Baxter, J., Bartlett, P.: Direct gradient-based reinforcement learning (1999)
19. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In Kremer, S.C., Kolen, J.F., eds.: A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press (2001)
20. Schmidhuber, J.: RNN overview (2004) http://www.idsia.ch/˜juergen/rnn.html.
21. Wieland, A.: Evolving neural network controllers for unstable systems. In: Proceedings of the International Joint Conference on Neural Networks (Seattle, WA), Piscataway, NJ: IEEE (1991) 667–673
22. Torcs: Torcs, the open racing car simulator. (2007) http://torcs.sourceforge.net/.