Recurrent Policy Gradients

Daan Wierstra [a,*], Alexander Förster [a], Jan Peters [b]
Jürgen Schmidhuber [a,c,d],

[a] *IDSIA, Manno–Lugano, Switzerland*
[b] *Max Planck Institute for Biological Cybernetics, Tübingen, Germany*
[c] *TU München, Institut für Informatik, Garching bei München, Germany*
[d] *University of Lugano, Faculty of Informatics, Lugano, Switzerland*

Abstract

Reinforcement learning for partially observable Markov decision problems (POMDPs) is a challenge as it requires policies with an internal state. Traditional approaches suffer significantly from this shortcoming and usually make strong assumptions on the problem domain such as perfect system models, state-estimators and a Markovian hidden system. Recurrent neural networks (RNNs) offer a natural framework for dealing with policy learning using hidden state and require only few limiting assumptions. As they can be trained well using gradient descent, they are suited for policy gradient approaches.

In this paper, we present a policy gradient method, the *Recurrent Policy Gradient* which constitutes a model-free reinforcement learning method. It is aimed at training limited-memory stochastic policies on problems which require long-term memories of past observations. The approach involves approximating a policy gradient for a recurrent neural network by backpropagating return-weighted characteristic eligibilities through time. Using a "Long Short-Term Memory" RNN architecture, we are able to outperform previous RL methods on three important benchmark tasks. Furthermore, we show that using history-dependent baselines helps reducing estimation variance significantly, thus enabling our approach to tackle more challenging, highly stochastic environments.

Key words: Recurrent Neural Networks, Policy Gradient Methods, Reinforcement Learning, Partially Observable Markov Decision Problems (POMDPs)

---

* Corresponding author.
Email addresses: `daan@idsia.ch` (Daan Wierstra), `alexander@idsia.ch` (Alexander Förster), `mail@jan-peters.net` (Jan Peters), `juergen@idsia.ch` (Jürgen Schmidhuber).

1 Introduction

Reinforcement learning (RL) is one of the most important problems in machine learning, psychology, optimal control and robotics [1]. In this setting, it is generally assumed that we have an agent that learns from trial and error, directly interacting with the environment to discover incrementally better policies. Despite all the successes in reinforcement learning, many common methods are limited to fully observable problems with no hidden states. However, RL tasks in realistic environments typically need to deal with incomplete and noisy state information resulting from partial observability such as encountered in partially observable Markov decision problems (POMDPs). Furthermore, the goal of dealing with non-Markovian problems is most likely beyond the abilities of traditional value function approaches. For such partially observable and non-Markovian problems, the optimal solution will require a policy representation with an internal memory. Among all function approximators with internal state, recurrent neural networks (RNN) appear to be the method of choice and can make a big difference in reinforcement learning problems. However, only few reinforcement learning methods are theoretically sound when applied in conjunction with such function approximation, and catastrophic divergence of traditional methods can be shown in this context [2].

Policy gradient (PG) methods constitute an exception, as these allow for learning policies even with noisy state information [3], work in combination with function approximation [4,5], are compatible with policies that have internal memory [6], can naturally deal with continuous actions [7–9] and are guaranteed to converge at least to a local minimum. Furthermore, most successful algorithms for solving real world reinforcement learning tasks are applications of PG methods, see, e.g., [10–12,3,7,13,14] for an overview. Provided the choice of policy representation is powerful enough, PGs can tackle quite complex RL problems.

At this point, policy gradient-based reinforcement learning exhibits two major drawbacks from the perspective of recurrent neural networks, i.e., (i) the lack of scalability of policy gradient methods in the number of parameters, and (ii) the small number of algorithms that were developed specifically for recurrent neural network policies with large-scale memory. Most PG approaches have only been used to train policy representations with maximally a few dozen parameters, while RNNs can have thousands. Surprisingly, the obvious combination with standard backpropagation techniques has not been extensively investigated (a notable exception being the SRV algorithm [15,16], which was, however, solely applied to feedforward networks). In this paper, we address this shortcoming, and show how PGs can be naturally combined with backpropagation, and BackPropagation Through Time (BPTT) [17] in particular, to form a powerful RL algorithm capable of training complex neural networks

with large numbers of parameters.

Work on policy gradient methods with memory has been scarce so far, largely limited to finite state controllers. Strikingly, memory models based on finite state controllers perform less than satisfactorily, even on quite simple benchmarks (e.g. single pole balancing without velocity information cannot be learned beyond 1000 time steps [6,18], whereas evolutionary methods and the algorithm presented in this paper manage to balance the pole $100,000+$ steps). We conjecture that the reason is that for finite state controllers a *stochastic* memory state model must be learned in conjunction with a policy, which is prohibitively expensive. In this paper, we extend policy gradient methods to more sophisticated policy representations capable of representing memory using an RNN architecture called Long Short-Term Memory (LSTM) for representing our policy [19]. We develop a new reinforcement learning algorithm aimed specifically at RNNs that can effectively learn memory-based policies for deep memory POMDPs. This algorithm, the *Recurrent* Policy Gradient (RPG) algorithm, backpropagates the estimated return-weighted *eligibilities* backwards through time using 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 three important RL benchmark tasks with different properties: continuous control in a non-Markovian double pole balancing environment, and discrete control on both the deep memory T-maze [20] task (which was designed to test an RL algorithm's ability to deal with extremely long term dependencies) and the still-unsolved (up to human-level performance) stochastic 89-state Maze task. Moreover, we show promising results in a complex car driving simulation which is challenging for humans. Here, we can show real-time improvement of the policy which has been largely unachieved in reinforcement learning for such complex tasks.

The paper is organized as follows. The next section describes the reinforcement learning framework and briefly reviews LSTM's architecture. The subsequent sections introduce the derivation of Recurrent Policy Gradient algorithm, and present our experimental results using RPGs with memory. The paper finishes with a discussion.

2 Preliminaries

In this section, we first briefly summarize the reinforcement learning terminology as used in this paper with a focus on RL for RNNs. Subsequently, we describe the particular type of recurrent neural network architecture used in this paper, i.e., Long Short-Term Memory networks.

3

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 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 $R_t = \sum_{k=t}^{\infty} r_k \gamma^{t-k}$ which is the *return* 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 $J = E[R_0] = \lim_{T \to \infty} \mathbf{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$. For the average reward case where $\gamma \to 1$ 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 $M(h_t)$ of the events which we call the limited memory of the agents past. Thus, a stochastic policy π can be defined as $\pi(a|h_t) = p(a|M(h_t); \theta)$, 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.
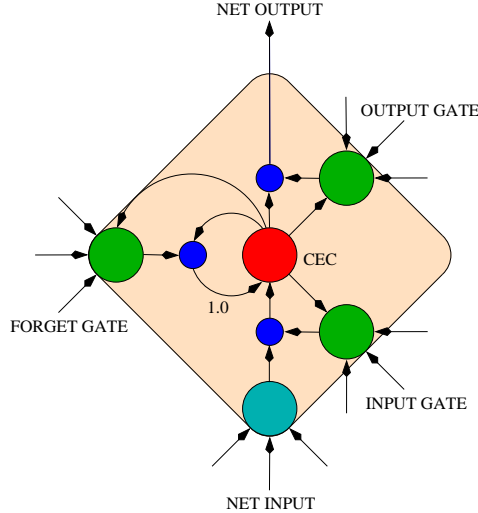
Fig. 1. The Long Short-Term Memory cell. The figure shows an LSTM cell with a net input, a Constant Error Carousel (CEC), an input gate, a forget gate and an output gate. The cell has an internal state CEC and a forget gate that determines how much the CEC is attenuated at each time step. The input gate controls access to the state by the external inputs and the outputs of other cells, and the output gate determines how much and when the cell fires.

2.2 LSTM Recurrent Neural Networks as Policy Representation

Recurrent neural networks are designed to deal with issues of time, such as approximating time series. A crucial feature of this class of architectures is that they are capable of relating events in a sequence, in principle even if placed arbitrarily far apart. A typical RNN π maintains an *internal state $M(h_t)$* (or *memory*) which it uses to pass on (compressed) history information to the next moment by using recurrent connections. At every time step, the RNN takes an input vector o_t and produces an output vector $\pi(M(h_t))$ from its internal state, and since the internal state $M(h_t)$ of any step is a function f of the previous state and the current input signal $M(h_t) = f(o_t, M(h_{t-1}); \theta)$, it can take into account the entire history of past observations by using its recurrent connections for recalling events. Not only can RNNs represent memory, they can, in theory, be used to model any dynamic system [21]. Like conventional neural networks, they can be trained using a special variant of backpropagation, backpropagation through time (BPTT) [17,22].

Usually BPTT is employed to find the gradient $\nabla_\theta E$ in parameters θ (that define f and π) for minimizing some error measure E, e.g. summed squared error. This is done by first executing a forward pass through the RNN all to the end of the sequence, at every time step unfolding the RNN, reusing parameters θ for the recurrent connections, producing outputs and computing the error δ_t. Then a (reverse) backwards pass is performed, computing the gradient backwards through time by backpropagating the errors. Usually, this is done

in a *supervised* fashion, but we will apply this technique to a reinforcement learning setting.

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* [23,24], 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 [19,25]), 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 differentiable *memory cells*, which can be used to store activations arbitrarily long. Access to the internal state of the memory cell (the Constant Error Carousel or CEC) is *gated* by gating units that learn to open or close depending on the context. Three types of (sigmoidal) gates are present: *input gates* that determine the input to the memory cell, *forget gates* that control how much of the CEC's value is transferred to the next time step, and *output gates* which regulate the output of the memory cell by gating the cell's output. See Figure 1 for a depiction of LSTM's structure.

LSTM networks have been shown to outperform other RNNs on numerous time series requiring the use of deep memory [26]. 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.

3   Recurrent Policy Gradients

In this section, we first formally derive the Recurrent Policy Gradient framework. Subsequently, history-dependent baselines are introduced, and the section is concluded with a description of the Recurrent Policy Gradient algorithm.

3.1 Derivation of Recurrent Policy Gradients

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. $R(H)$ could be the average of the rewards for the average reward case, or the discounted sum for the discounted case. Let $p(H|\theta)$ denote the probability of a history given policy-defining weights θ. 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 policy π. In order to be able to apply gradient ascent to find a better policy, we have to find the gradient $\nabla_\theta J$, which can then be used to incrementally update parameters θ of policy π in small steps. Since we know that rewards $R(H)$ for a given history H do *not* depend on the policy parameters θ (that is, $\nabla_\theta R(H) = 0$), we can write $\nabla_\theta J = \nabla_\theta \int_H p(H|\theta)R(H)dH = \int_H \nabla_\theta p(H|\theta)R(H)dH$. Now, using the "likelihood-ratio trick" we find

$$
\begin{aligned}
\nabla_\theta J &= \int \nabla_\theta p(H)R(H)dH \\
&= \int \frac{p(H)}{p(H)}\nabla_\theta p(H)R(H)dH \\
&= \int p(H)\nabla_\theta \log p(H)R(H)dH.
\end{aligned}
$$

Taking the sample average as Monte Carlo (MC) approximation of this expectation by taking N trial histories we get

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

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 [7] 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.

3.2 History-dependent Baselines

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 $\sigma_T^2 = \mathbf{E}[\sum_{t=0}^{T} (\nabla_\theta \log \pi(a_t | h_t^n))^2]$ which grows linearly with T. One way to tackle such problems and reduce this variance is to include a constant *baseline* b (first introduced by Williams [27]) into the gradient estimate $\nabla_\theta J \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi(a_t | h_t^n)(R_t^n - b)$. Baseline b is typically taken to be the expected average return and subtracted from the actual return, such that the resulting quantity $(R_t - b)$ intuitively yields information on whether the return was better or worse than expected. 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 $\mathbf{E}[\sum_{n=1}^{N} \nabla_\theta \log p(H_t^n) b] = 0$ and, thus, the baseline can only reduce the variance but not bias the gradient in any way [27].

Whereas previously a *constant* baseline was used, we can in fact extend the baseline concept to include subhistory-dependent function approximators $B(h_t)$ parameterized by w. The correctness of this approach can be realized by applying the same trick $\int_a \nabla_\theta \pi(a | h_t) B(h_t) da = 0$ for every possible subhistory h_t. Now the baseline $B(h_t)$ can be represented as an LSTM RNN

receiving observations and actions as inputs, trained to predict future return given the current policy π. This construct closely resembles the concept of value functions in temporal difference methods. However, note that we do not use temporal difference methods for training the history-dependent baseline network (since such updates can be arbitrarily bad in partially observable environments [2]), but apply supervised training using simply the actually experienced returns as targets for every time step. Using non-constant, history-dependent baselines, our algorithm now uses *two* separate RNNs: one policy π parameterized by θ, and one baseline network B parameterized by w. Using the extended baseline network, the gradient update for the policy now becomes $\nabla_\theta J \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi(a_t | h_t^n)(R_t^n - B(h_t^n))$.

3.3 The Recurrent Policy Gradients Algorithm

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 [17]) 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 [27] 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 done in [6,18] – eligibility backpropagation through time disambiguates relevant hidden state automatically, when possible.

4 Experiments

We carried out experiments on four 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. RPGs outperform all other single-agent methods on this task, as far as we are aware. The second task, the T-maze, is a difficult discrete control task that requires remembering its initial observation until the end of the episode. On this task, RPGs outperformed the second-best method by more than an order of magnitude for longer corridors. The third task, the 89-state Maze, is a highly stochastic POMDP maze task which has yet to be solved up to human level performance. On this task, RPGs outperform all other (model-free) algorithms.

Last, we show promising results on a complex car driving simulation (TORCS) which is challenging for humans. Here, we can show real-time improvement of the policy, something which has been largely unachieved in reinforcement learning for such complex tasks.

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, except for the 89-state Maze task, where an additional LSTM network was used to estimate a history-dependent baseline.

4.1 Continuous Control: Partially Observable Single & Double Pole Balancing

This task involves trying to balance a pole hinged on a cart that moves on a finite track (see Figure 2). 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 β and velocities $\dot{x}$ and $\dot{\beta}$. 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

10
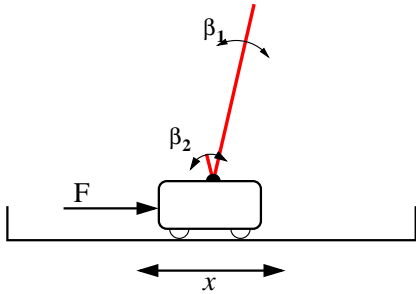
	Markov	non-Markov
1 pole	863 ± 213	1893 ± 527
2 poles	4981 ± 1386	5649 ± 1548

Fig. 2. The non-Markov double pole balancing task. The task consists of a moving cart on a track, with two poles of different lengths ($1m$ and $0.1m$) hinged on top. The controller applies a (continuous) force F to the cart at every time step, after observing pole angles β_1 and β_2. The objective is to indefinitely keep the poles from falling. The table shows the results for RPGs on the pole balancing task, for the four possible cases investigated in this paper: 1 pole Markov, 2 poles Markov, 1 pole non-Markov, and 2 poles non-Markov. The results show the mean and standard deviation of the number of evaluations until the success criterion was reached, that is, when a run lasts more than $10,000$ time steps. Results are computed over 20 runs.

task more challenging, we (1) remove velocity information $\dot{x}$ and $\dot{\beta}$ 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 [28], 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 for our LSTM RNN, consisting of two output neurons: a mean μ (which was interpreted linearly) and a standard deviation σ (which was scaled with the logistic function between 0 and 1 in order to prevent variances from being negative) where eligibilities were calculated according to [27]. We use a learning rate $\alpha\sigma^2$ (as suggested by Williams [27]) to prevent numerical instabilities when variances tend to 0, and use learning rate $\alpha = 0.001$, $momentum = 0.9$ and discount factor $\gamma = 0.99$. 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 2 shows results averaged over 20 runs. RPGs clearly outperform earlier PG methods (for a comparison, see [18]'s finite state controller, which cannot balance a single pole in a partially observable setting for more than 1000 time steps, even after 500,000 trials). As far as we are aware, RPGs constitute the only published single-agent approach that can satisfactorily solve this problem.
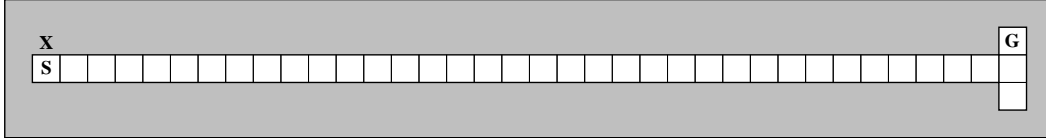
Fig. 3. The T-maze task. The agent observes its immediate surroundings and is capable of the actions goNorth, goEast, goSouth, and goWest. It starts in the position labeled 'S', there and only there observing either the signal 'up' or 'down', indicating whether it should go up or down at the end of the alley. It receives a reward if it goes in the correct direction, and a punishment if not. In this example, the direction is 'up' and N, the length of the alley, is 35.

4.2 Reinforcement Learning in Discrete POMDPs

In this section, we show the high performance of our algorithm for traditional discrete POMDP problems. The *Long Term Dependency T-maze* from Section 4.2.1 is a standard benchmark for learning deep-memory POMDPs while *the 89-state Maze* in Section 4.2.2 is a problem where humans are still able to beat the best known algorithmically learned policy (e.g. see [20]).

4.2.1 Long Term Dependency T-maze

The second experiment was carried out on the T-maze [20] (see Figure 3). 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 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 3 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 4 the results are displayed, in addition to other algorithms' results (RL-Elman and RL-LSTM) taken from [20], of which the results on RL-LSTM were the best results reported so far. We can see that RPGs clearly outperform the value-based methods, even by more than an order of magnitude in terms
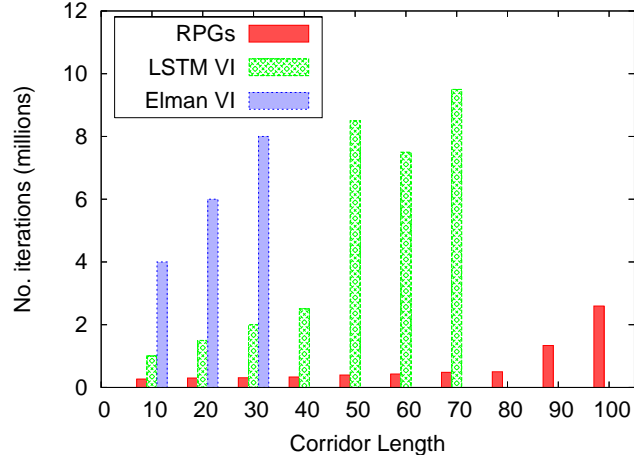
12

Fig. 4. T-maze results. Elman-based Value Iteration (Elman VI) starts to degrade after corridor length $N = 10$, LSTM Value Iteration (LSTM VI) falters after $N = 50$, while Recurrent Policy Gradients' performance starts to degrade at length $N = 100$. The 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 [20]).

of iterations for corridor lengths longer than 40. Additionally, RPGs are able to solve this task up to N=90, while the second best algorithm, RL-LSTM, solves it up to N=50. The large performance gain on this task for Recurrent Policy Gradients might be due to the difference in complexity of learning a simple (memory-based) policy versus learning unnecessarily complex value functions. Nevertheless, the impressive performance advantage of RPGs over value-based methods on this domains indicates a possibly significant potential for the application of Recurrent Policy Gradients to other deep-memory domains.

4.2.2 The 89-state Maze

In this extremely stochastic benchmark task (see Figure 5; see [29] for a complete description) the aim for the agent is to get to the goal as fast as possible (where the reward is 1, other locations have reward 0) from a random starting position and orientation, but within 251 time steps. For reward attribution, discount factor $\gamma = 0.98$ is used. The agent has not only a position, but also an orientation, and its actions consist of moving forward, turning left, turning right, turning about, and doing nothing. State transactions are extremely noisy. Observations, which consist of 4 bits representing adjacent wall information (wall or no wall), are noisy and are inverted with probability 10%, which sets the chance of getting the correct observation somewhere between 0.65 and 0.81, depending on the agent's location. It is interesting to note that, to the authors' knowledge, this domain has as of yet not been satisfactorily
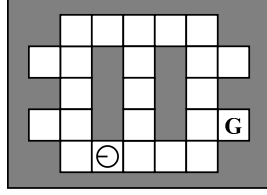
Fig. 5. The 89-state maze. In this extremely stochastic maze, the agent has a position, an orientation, and can execute five different actions: forward, turnleft, turnright, turnabout, and doNothing. The agent starts every trial in a random position. Its goal is to move to the square labeled 'G'. Observations comprise the local walls but are noisy (there is a high probability of observing walls where there are none and vice versa). Action outcomes are noisy and cannot be relied on. See [29] for a complete description of this problem domain.

solved, that is, solved up to human-comparable performance. Humans still greatly outperform all algorithms we are aware of. That is what makes this a very interesting and challenging task.

Because of the random starting position, this task is extremely difficult without the use of any history-dependent baseline, since the agent might start close to the target or not, which influences the expected rewards accordingly. That is why we apply a history-dependent baseline for this task, trained with $\alpha = 0.001$ and $momentum = 0.9$ after every episode. 20 runs were performed to test the performance of the algorithm, using a history-dependent baseline which was trained on actually received returns using a separate LSTM network with 10 memory cells with the same inputs as the policy network including a bias. Each run was executed for $30,000,000$ iterations. After that, the resulting policy was evaluated. The median number of steps to achieve the goal (in case the goal is achieved) was 58, and the goal was reached in 95% of the trials. This compares favorably with the second best other (model-free) method the authors are aware of, Bakker's RL-LSTM algorithm [30] with 61 steps and 93.9%, respectively. In [29] the human performance of 29 steps and 100% is highlighted, which again underlines the difficulty of the task. However, the fact that RPGs outperform all other algorithms on this task might indicate that the application of Recurrent Policy Gradients to RNNs, especially in combination with history-dependent baselines, might indeed be fruitful.

4.3   Car Racing with Recurrent Policy Gradients

In order to show that our algorithm performs well in a complicated real task which is difficult for humans, we have carried out experiments on the TORCS [31] car racing simulator. TORCS is an advanced open source racing game with a graphical user interface and simulated simplified physics which provide a challenging experience for game play. Additionally to being open

source, the game was specifically designed for programming competitions between steering agents, and the code framework allows for easy plug-ins of code snippets for competitions between preprogrammed drivers. As such, it provides a perfect testbed for reinforcement learning algorithms that aim to go beyond the current benchmark standards.

We trained our RPG agent on one single track (see Figure 6), on which it has to learn to drive a Porsche GT1 and stay on the road while achieving high speed. Whenever the car gets stuck off the road, a learning episode ends, the car is put back on track and a new episode begins. The steering outputs of the RNN, which were executed at a rate of 30 frames per second, are interpreted as a Gaussian with one output neuron interpreted linearly (μ, the mean) and one output neuron interpreted logistically between 0 and 1 (σ, the standard deviation) to ensure it is nonnegative. The four observations which were normalized around 0 with std 1, include a bias, the speed, the steering angle, position on the road and look-ahead-distance (which was linearly varied with speed). Its rewards consist of speed measurements at every time step, and the agent receives negative rewards for spending time off track. A large penalty is inflicted upon the car getting stuck off track, which ends an episode. The car's speed starts off at 10 km/h, which is gradually increased over time to reach 70 km/h after 30 minutes.

We performed 10 runs on this lap using the same learning settings and 10-cell network as applied to the non-Markovian double pole balancing task. The baseline was updated after every 100 time steps. We found that the agent learns, for all runs, to consistently steer and stay on the road after just under 2 minutes of real-time behavior. In all runs, the car first drives off the track immediately four or five times, then learns to stay on track until it hits the first curve, where it slides off again. Within two minutes, however, it drives nearly perfectly in the middle off the road, and learns to 'cut curves' slightly when the speed is increased gradually to 70 km/h after 30 minutes. The agent can learn to drive safely – not getting off track – up to 70 km/h, after which its behavior destabilized in all runs. Future work will investigate how to make the behavior more robust and how to cope with higher speeds. This will have to include speed control and braking by the network as well, which could be actualized using additional (softmax) output neurons for gears, brakes and gas. The fastest lap time achieved after 30 minutes of training was just under 3 minutes, which is, unfortunately, still twice as slow as a trained human player or our preprogrammed agent.

To conclude, our car driving agent learns fast, in real-time (2 minutes), to steer correctly and keep the vehicle on the road. This is about as fast as a novice human player learns to stay on the road. Moreover, it reaches high speeds of up to 70 km/h within 30 minutes of online training time. Although rigid preprogrammed speed control destabilizes the agent with higher speeds,

Fig. 6. The TORCS racing car simulator.

the fast learning suggests this approach might be worth investigating when dealing with real-time learning problems in continuous robot control.

5 Conclusion

In this paper, we have introduced Recurrent Policy Gradients, an elegant and powerful method for dealing with reinforcing learning in partially observable environments. The algorithm, an RNN-based policy gradient method equipped with memory capable of memorizing events from arbitrarily far in the past, involves computing and backpropagating action eligibilities through time with 'Long Short-Term Memory' memory cells, thus updating a policy which maps event histories to action probabilities. The approach outperformed other RL methods on three important benchmarks with different characteristics. We think Recurrent Policy Gradients might constitute both one of the simplest, and one of the most efficient RL algorithms to date for difficult non-Markovian tasks.

Acknowledgments

References

[1] R. Sutton, A. Barto, Reinforcement learning: An introduction, Cambridge, MA, MIT Press, 1998.

[2] S. P. Singh, T. Jaakkola, M. I. Jordan, Learning without state-estimation in partially observable Markovian decision processes, in: International Conference

on Machine Learning (ICML 1994), Morgan Kaufmann, San Francisco, CA, 1994, pp. 284–292.

[3] J. Baxter, P. Bartlett, L. Weaver, Experiments with infinite-horizon, policy-gradient estimation, Journal of Artificial Intelligence Research 15 (2001) 351–381.

[4] R. Sutton, D. McAllester, S. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in: Advances in Neural Information Processing Systems (NIPS), MIT Press, 2001, pp. 1057–1063.

[5] S. Bhatnagar, R. Sutton, M. Ghavamzadeh, M. Lee, Incremental natural actor-critic algorithms., in: Advances in Neural Information Processing Systems (NIPS), MIT Press, 2007, pp. 105–112.

[6] D. Aberdeen, Policy-gradient algorithms for partially observable Markov decision processes, Ph.D. thesis, Australian National University, Australia (2003).

[7] J. Peters, S. Schaal, Policy gradient methods for robotics, in: Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS), Beijing, China, 2006, pp. 2219 – 2225.

[8] N. Kohl, P. Stone, Policy gradient reinforcement learning for fast quadrupedal locomotion, in: Proceedings of the IEEE International Conference on Robotics and Automation, Vol. 3, 2004, pp. 2619–2624.

[9] J. Peters, S. Schaal, Reinforcement learning of motor skills with policy gradients, Neural Networks 21 (4) (2008) 682–97.

[10] H. Benbrahim, J. Franklin, Biped dynamic walking using reinforcement learning, Robotics and Autonomous Systems 22 (3–4) (1997) 283–302.

[11] J. Moody, M. Saffell, Learning to Trade via Direct Reinforcement, IEEE Transactions on Neural Networks 12 (4) (2001) 875–889.

[12] D. Prokhorov, 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, pp. 268–271.

[13] J. Peters, S. Vijayakumar, S. Schaal, Natural Actor Critic, in: Proceedings of the 16th European Conference on Machine Learning (ECML 2005), 2005, pp. 280–291.

[14] J. Peters, S. Schaal, Natural Actor Critic, Neurocomputing 71 (7–9) (2008) 1180–1190.

[15] V. Gullapalli, A stochastic reinforcement learning algorithm for learning real-valued functions, Neural Networks 3 (6) (1990) 671–692.

[16] V. Gullapalli, Reinforcement learning and its application to control, Ph.D. thesis, University of Massachusetts, Amherst, MA., USA (1992).

[17] P. Werbos, Back propagation through time: What it does and how to do it., in: Proceedings of the IEEE, Vol. 78, 1990, pp. 1550–1560.

[18] N. Meuleau, L. Peshkin, K.-E. Kim, L. P. Kaelbling, Learning finite-state controllers for partially observable environments, in: Proc. Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI '99). Morgan Kaufmann, 1999, pp. 427–436.

[19] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (8) (1997) 1735–1780.

[20] B. Bakker, Reinforcement learning with Long Short-Term Memory, in: Advances in Neural Information Processing Systems 14, MIT Press, 2002, pp. 1475–1482.

[21] H. T. Siegelmann, E. D. Sontag, Turing computability with neural nets, Applied Mathematics Letters 4 (6) (1991) 77–80.

[22] R. J. Williams, D. Zipser, A learning algorithm for continually running fully recurrent networks, Neural Computation 1 (2) (1989) 270–280.

[23] Y. Bengio, P. Simard, P. Frasconi, Learning long-term dependencies with gradient descent is difficult, IEEE Transactions on Neural Networks 5 (2) (1994) 157–166.

[24] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, in: S. C. Kremer, J. F. Kolen (Eds.), A Field Guide to Dynamical Recurrent Neural Networks, IEEE Press, 2001, pp. 237–244.

[25] F. A. Gers, N. Schraudolph, J. Schmidhuber, Learning precise timing with LSTM recurrent networks, Journal of Machine Learning Research 3 (2002) 115–143.

[26] J. Schmidhuber, RNN overview, http://www.idsia.ch/˜juergen/rnn.html (2004).

[27] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Machine Learning 8 (1992) 229–256.

[28] A. Wieland, Evolving neural network controllers for unstable systems, in: Proceedings of the International Joint Conference on Neural Networks (Seattle, WA), Piscataway, NJ: IEEE, 1991, pp. 667–673.

[29] M. Littman, A. Cassandra, L. Kaelbling, Learning policies for partially observable environments: Scaling up, in: A. Prieditis, S. Russell (Eds.), Machine Learning: Proceedings of the Twelfth International Conference, Morgan Kaufmann Publishers, San Francisco, CA, 1995, pp. 362–370.

[30] B. Bakker, The state of mind: : Reinforcement learning with recurrent neural networks, Ph.D. thesis, Leiden University, the Netherlands (2004).

[31] Torcs, The open racing car simulator., http://torcs.sourceforge.net/ (2007).