Policy Gradient Critics

Daan Wierstra[1] and Jürgen Schmidhuber[1,2]

[1] Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA),
CH-6928 Manno-Lugano, Switzerland, daan@idsia.ch
[2] Department of Embedded Systems and Robotics, Technical University Munich,
D-85748 Garching, Germany, schmidhu@in.tum.de

Abstract. We present Policy Gradient Actor-Critic (PGAC), a new model-free Reinforcement Learning (RL) method for creating *limited-memory stochastic policies* for Partially Observable Markov Decision Processes (POMDPs) that require long-term memories of past observations and actions. The approach involves estimating a policy gradient for an Actor through a Policy Gradient Critic which evaluates probability distributions on actions. Gradient-based updates of history-conditional action probability distributions enable the algorithm to learn a mapping from memory states (or event histories) to probability distributions on actions, solving POMDPs through a combination of memory and stochasticity. This goes beyond previous approaches to learning purely reactive POMDP policies, without giving up their advantages. Preliminary results on important benchmark tasks show that our approach can in principle be used as a general purpose POMDP algorithm that solves RL problems in both continuous and discrete action domains.

1 Introduction

Reinforcement Learning [1] algorithms often need to deal with partial observability problems naturally arising in real-world tasks. A naive approach would be to learn *reactive stochastic policies* [2] which simply map observations to probabilities for actions. The underlying philosophy here is that utilizing random actions, as opposed to deterministic actions, will prevent the agent from getting stuck. In general this is clearly suboptimal, and the employment of some form of memory seems essential for many realistic RL settings. However, for cases where our memory system's capacity is *limited* – that is, imperfect, like for example all Recurrent Neural Network architectures – reactive stochasticity may still facilitate learning good policies for realistic environments. Hence the need to stress the importance of learning what we define as *limited-memory stochastic policies*, that is, policies that map limited memory states to probability distributions on actions.

In spite of their apparent advantages, work on limited-memory stochastic policies has been scarce so far, notable exceptions being finite state-based policy gradients [3, 4] and evolutionary search (e.g. [5]). We propose a novel approach to learning limited-memory stochastic policies in partially observable environments. Conventional policy gradient approaches update policy parameters using a sampling-based Monte Carlo estimate of the gradient in policy space – they constitute a framework of Actor-only methods – but this tends to lead to high variance estimates. Our new approach Policy Gradient Actor-Critic (PGAC), however, is a dual Actor-Critic [1] architecture and updates

a policy's parameters – the Actor – using a *model-based* estimated gradient on action probabilities, the model being the Policy Gradient Critic. Since PGAC uses a gradient that is provided directly by a Policy Gradient Critic, representing an action distribution evaluation function over pairs of history / action distribution parameters, we can avoid brute force Monte Carlo sampling, which provides numerous advantages including the power of function approximator generalization. Computing the gradient from the Policy Gradient Critic has the potential to yield a substantially improved estimate of beneficial policy updates for the Actor. Moreover, by representing a Q-function over action probability distributions rather than over concrete actions, we can explicitly represent the value of stochasticity.

We use Long Short-Term Memory (LSTM) [6] as our memory-capable differentiable recurrent function approximator (DRFA) for both Actor and Policy Gradient Critic. DRFA-based RL algorithms are a perfect example of limited-memory algorithms. The recurrency or *memory* of DRFAs can capture hidden state effects, which enables the algorithm to deal with the partial observability that plagues many real-world tasks. Like other DRFAs, they are limited in their learning capacity, however, and prone to local minima. These limitations can be partly compensated for by using PGAC's explicitly learned value of stochasticity in action probabilities. We show that the resulting PGAC method using LSTM is able to solve three fundamentally different benchmark tasks: continuous control in a non-Markovian pole balancing task, discrete control on the *deep memory* T-maze task, and discrete control on the extremely stochastic 89-state Maze.

2 The Algorithm

In this section we explain our basic algorithm for learning limited-memory stochastic policies. First, we summarily review the Reinforcement Learning framework as used in this paper. The particular differentiable recurrent function approximator applied, LSTM, is briefly described. Then we outline how PGAC operates within the RL framework using LSTM as DRFA for both Actor and Policy Gradient Critic.

2.1 Reinforcement Learning – Generalized Problem Statement

First let us introduce the notation used in this paper and the corresponding RL framework (see Figure 1 for a schematic depiction of the framework used). The environment, which is assumed to be Markovian, produces a state g_t at every time step. Transitions from state to state are governed by probabilities $T(g_{t+1}|a_t, g_t)$ dependent upon action a_t executed by the agent. 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. Both quantities, governed by fixed distributions $p(o|g)$ and $p(r|g)$, are solely dependent on the state g_t. Let $R_t = \sum_{k=t+1}^{\tau} r_k \gamma^{t-k-1}$ be the *return* at time t, where $0 < \gamma < 1$ denotes a discount factor and τ denotes the length of the episode. The agent operates in episodes 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 only on g_t. The environment-generated experience consists of finite episodes. A *history* h_t is the string
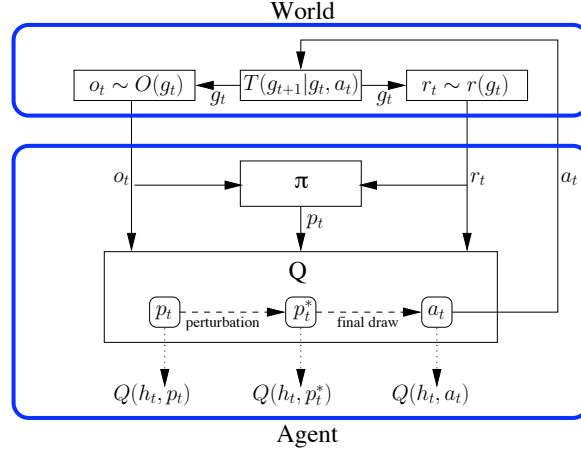
Fig. 1. PGAC in the reinforcement learning framework. Shown is the interaction between the Agent and the World. The Agent consists of two components, the Actor π_θ parameterized by weights θ and the Policy Gradient Critic Q_w parameterized by weights w. Q evaluates Actor-provided action probability distribution parameters $\mathbf{p}$, its perturbation $\mathbf{p}^*$ and the actually executed action a. Using the Jacobian of the Policy Gradient Critic, the parameters/weights of the DRFA defining the Actor are updated using regular gradient ascent.

of observations, rewards and actions up to moment t since the beginning of the episode: $h_t = \langle o_0, r_0, a_0, o_1, r_1, a_1, \ldots, o_t, r_t \rangle$. Policy $\pi(s_t; \theta)$ produces a vector $\mathbf{p}_t$ of probability distribution parameters (describing $\mathcal{D}_{\mathbf{p}_t}$) over actions given a memory state, from which actions a_t are drawn $a_t \sim \mathcal{D}_{\mathbf{p}_t}$. Here, internal state variable s is some DRFA-trained representation of the agent's entire history of actions, rewards and observations $s_t = f^*(\langle o_0, r_0, a_0, o_1, r_1, a_1, \ldots, o_t, r_t \rangle; \theta) = f^*(h_t; \theta) = f(s_{t-1}, (o_t, r_t); \theta)$. Now the objective of our algorithm is to optimize expected future discounted reward $\mathbf{E}[R_1] = \mathbf{E}\left[\sum_{t=1}^{\tau} \gamma^t r_t\right]$ by adjusting action probabilities $\mathcal{D}_{\mathbf{p}_t}$ appropriately. The estimate on the quality of action probabilities described by some parameter vector $\mathbf{p}$ can be expressed by a Q-function-like quantity: $Q(h_t, \mathbf{p}) = Q(f(h_t), \mathbf{p}) = Q(s_t, \mathbf{p})$ is a function that indicates the estimated expected future discounted reward of choosing actions following distribution $\mathcal{D}_\mathbf{p}$ at time step t after history h_t, *and following policy π thereafter*: $Q(h_t, \mathbf{p}) \approx \mathbf{E}\left[R_t | h_t, a_t \sim \mathcal{D}_{\mathbf{p}_t}, a_{k>t} \sim \mathcal{D}_\pi\right]$.

2.2 LSTM Recurrent Function Approximators

Differentiable recurrent function approximators constitute a class of architectures 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 DRFA π maintains an *internal state s_t* (its memory so to say) which it uses to pass on (compressed) history information to the next moment by using recurrent connections. At every time step, the DRFA

takes an input vector o_t and produces an output vector $\pi(s_t; \theta)$ from its internal state, and since the internal state s_t of any step is a function f of the previous state and the current input signal $s_t = f(o_t, s_{t-1}; \theta)$, it can take into account the entire history of past observations by using its recurrent connections for recalling events. Like conventional neural networks, DRFAs can be trained using backpropagation and related techniques based on gradient information. However, backpropagation is modified such that it works through time (BackPropagation Through Time (BPTT) [7, 8]).

Recurrent Neural Networks (RNNs), a subset of this class of algorithms, 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* [9], the fact that the gradient 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 circumvent this problem is Long Short-Term Memory (LSTM), 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 – context being present observations o_t and the previous internal state s_{t-1}.

LSTM has been shown to outperform other RNNs on time series requiring the use of deep memory [10]. Therefore, they seem well-suited for usage in RL algorithms for complex, deep memory requiring tasks. Whereas DRFAs are typically used for next step prediction, we use them as a function approximator to both estimate *value change* (the Policy Gradient Critic) and to *control* (the Actor) given histories of observations, actions and rewards.

2.3 Policy Gradient Actor-Critic

PGAC Reinforcement Learning for stochastic policies relies on the following observation: actions can be represented as special cases of *probability distribution parameters*. For example, any discrete action a can be represented as a special vector $\mathbf{p}$ where one element of $\mathbf{p}$ is 1 and the other 0. Action a_2 in a three-dimensional discrete action space can be expressed $\mathbf{p} = [0, 1, 0]$. We can apply this representation to conventional value functions, but now we can express more. Representing actions as probability distribution parameters enables us to construct Q-value functions over *action probabilities*. For example, $Q(s, [0.5, 0.5])$ would denote the value of executing a_1 with probability 0.5 and executing a_2 with probability 0.5 in state s. For a one-dimensional Gaussian case, a single action could be represented as $\mathbf{p} = [\mu, \sigma] = [3.0, 0.0]$ with $\mu = 3.0$ and $\sigma = 0.0$, but now this vector is more expressive: $Q(s, [\mu, \sigma])$ represents the estimated expected value of executing a continuous action a drawn from normal distribution $a \sim \mathcal{N}(\mu, \sigma^2)$.

Like many conventional temporal difference learning algorithms for POMDPs, the PGAC algorithm uses two differentiable recurrent function approximators: one Actor π_θ parameterized by θ, and one Critic Q_w parameterized by w. The crucial difference between PGAC and other methods is the fact that its Policy Gradient Critic's Q-function

---

Algorithm 1 Policy Gradient Actor-Critic

---
for each episode e **do**
    for each time step t **do**
        Actor produces parameter vector $\mathbf{p}_t$ from $\pi(h_t; \theta)$
        Perturb vector $\mathbf{p}_t$ to $\mathbf{p}_t^*$: $\mathbf{p}_t^* \sim \mathcal{P}(\mathbf{p}_t)$
        Finally draw an action a_t according to $\mathbf{p}_t^*$: $a_t \sim \mathcal{D}_{\mathbf{p}_t^*}$
        Execute action a_t, observe effects o_{t+1} and r_{t+1}
        Update (SARSA-fashion) the Policy Gradient Critic's Q_w-value function for $\langle h_t, \{\mathbf{a}_t, \mathbf{p}_t, \mathbf{p}_t^*\} \rangle$ pairs using the following TD-errors for updating w:

$$E^{TD}\langle h_{t-1}, \mathbf{p}_{t-1} \rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{p}_{t-1})$$
$$E^{TD}\langle h_{t-1}, \mathbf{p}_{t-1}^* \rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{p}_{t-1}^*)$$
$$E^{TD}\langle h_{t-1}, \mathbf{a}_{t-1} \rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{a}_{t-1})$$

    Update Actor's parameters θ defining policy π as

$$\Delta\theta = \sum_i \alpha \overbrace{\frac{\partial Q(h_t, \mathbf{p}_t; w)}{\partial \mathbf{p}_t^{(i)}}}^{Critic} \underbrace{\frac{\partial \mathbf{p}_t^{(i)}}{\partial \theta}}_{Actor}$$

    end for
  end for

---

evaluates probability distributions over actions rather than single actions. The Policy Gradient Critic is forced to operate on incomplete information, i.e. it has to be able to provide estimates on the quality of the policy given that the agent intends to let the actual action be drawn from a probability distribution governed by $\mathbf{p}$. This way, the agent explicitly includes the value of stochasticity in action selection in its Q-function. The fact that this extended Policy Gradient Critic evaluates action probability distributions, combined with the fact that the Actor provides the parameters for such a distribution, eliminates the need for a prewired exploration policy, since exploration is adjusted online while executing the policy. Another important reason for explicitly representing the value of stochasticity is that, because of limited memory, a stochastic policy might be the optimal one for some real-world tasks. It has been shown [2] that for some domains deterministic policies can produce arbitrarily worse performance than stochastic policies.

The Actor π_θ outputs, at every time step t, deterministically given history h_t, probability distribution parameters $\mathbf{p_t} = \pi(h_t; \theta)$ from which the agent's actions are drawn $a_t \sim \mathcal{D}_{p_t}$. Additionally, the Policy Gradient Critic $Q(h_t, \mathbf{p}; w)$ estimates the value $\mathbf{E}\left[R_t | h_t, a_t \sim \mathcal{D}_\mathbf{p}, a_{k>t} \sim \mathcal{D}_\pi\right]$ for Actor-produced $\mathbf{p_t}$. After every episode, the Actor's policy can be updated by using the Policy Gradient Critic as a teacher which provides a derivative for the direction in which to adjust Actor-provided policy $\mathbf{p}_t = \pi(h_t; \theta)$. The Policy Gradient Critic can be taught using on-policy Temporal Difference Learning techniques.

Actor Learning The Actor is updated by updating Actor-defining parameters θ in the direction of higher expected future discounted reward *as predicted by the Policy Gradient Critic*:

$$\Delta\theta = \alpha \sum_i \frac{\partial Q(h_t, \mathbf{p}_t)}{\partial \mathbf{p}_t^{(i)}} \frac{\partial \mathbf{p}_t^{(i)}}{\partial \theta}$$

where $\mathbf{p}_t^{(i)}$ denotes parameter i of distribution parameter vector $\mathbf{p}_t = \pi(h_t)$ produced by the Actor at time t, and α denotes the learning rate. In order to compute this, using a fixed Policy Gradient Critic, we backpropagate a gradient signal towards higher Q-values from the Policy Gradient Critic's inputs $\mathbf{p}$, yielding the Jacobian of the Policy Gradient Critic, the quantities $\frac{\partial Q(h_t, \mathbf{p}_t^{(i)})}{\partial \mathbf{p}_t^{(i)}}$ for all i action probability distribution parameters. These values are then further backpropagated from the Policy Gradient Critic into the Actor architecture, now updating parameters θ along the way. In essence, the Policy Gradient Critic provides an estimate of the expected steepest gradient ascent for future discounted reward on the current incoming action distribution parameters. These estimated derivatives are then used by the Actor to update its policy, exactly in the direction of better value suggested by the Policy Gradient Critic. The algorithm pseudocode is provided in Algorithm 1.

Policy Gradient Critic Learning The Actor can only be updated if the Policy Gradient Critic provides sufficiently accurate estimations on future discounted reward. To train the Policy Gradient Critic, on-policy Temporal Difference Learning is used. Unlike most reinforcement learning algorithms, PGAC does not learn a Q-value for actions performed $Q(h, a; w)$, it rather learns a Q-value for distributions on actions: the Q-value $Q(h_t, \mathbf{p}; w)$, represented by the Policy Gradient Critic, learns the expected value of executing one action randomly drawn from probability distribution $\mathcal{D}_\mathbf{p}$, and following stochastic policy π_θ thereafter.

The Q-function estimates the value of a stochastic action under the policy provided by the Actor. The Temporal Difference (TD) Errors E^{TD} that can be easily extracted from the experience are (history, action probability distribution) pairs $\langle h_{t-1}, \mathbf{p}_{t-1} \rangle$ and $\langle h_{t-1}, \mathbf{a}_{t-1} \rangle$:

$$E^{TD}\langle h_{t-1}, \mathbf{p}_{t-1} \rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{p}_{t-1})$$
$$E^{TD}\langle h_{t-1}, \mathbf{a}_{t-1} \rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{a}_{t-1})$$

where $\mathbf{p}$ is the Actor-produced vector of action probability distribution parameters, and $\mathbf{a}$ denotes the actually executed action. The algorithm uses the Policy Gradient Critic's Jacobian to update the Actor, that is, it has to be able to represent how a *difference* in action probabilities relates to a *difference* in value. The above two TD-errors might not provide enough data points to reliably estimate such derivatives, though, since the region *around* $\mathbf{p}_t$ is not sampled by the Actor, although that is the region where the most useful training information is localized. Therefore, we want to add perturbed samples around $\mathbf{p}_t$ in order to be able to estimate how the Q-value changes with respect to $\mathbf{p}_t$.

Providing such samples *without biasing learning* can be done using what we call a 'perturbation / final draw' operation. A 'perturbation' operation $\mathcal{P}$ perturbs probability distribution parameters $\mathbf{p}$ – provided by the Actor – onto a new parameter vector $\mathbf{p}^* \sim \mathcal{P}(\mathbf{p})$, such that the expected distribution of actions a drawn from $a \sim \mathcal{D}_{\mathbf{p}^*}$ follow the same distribution as actions drawn from the original $a \sim \mathcal{D}_{\mathbf{p}}$.

Example distributions that can be perturbed are finite discrete distributions, where all elements of vector $\mathbf{p}$ sum up to 1 (which, in our experiments, is implemented as a softmax layer), or a Gaussian distribution, e.g. $\mathbf{p} = [\mu.\sigma]$. For finite discrete distributions, one way to construct a perturbation operation is to use a random number generator u_i where each u_i represents a uniformly distributed random number between 0 and 1. Good approximate values for $\mathbf{p}^*$ can then be generated by $p_i^* = \frac{p_i + \beta p_i u_i}{\sum_j p_j + \beta p_j u_j}$ where β is a constant, taken to be 1 in this paper. For the simple Gaussian case $\mathbf{p} = [\mu_p, \sigma_p]$, we could construct perturbation $\mathbf{p}^* = [\mu_{p^*} \sim \mathcal{N}(\mu_p, \sigma_p/2), 0.866\sigma_p]$.

Thus constructing $\mathbf{p}^*$ values around $\mathbf{p}$ provides us with informative extra samples – we could see them as hypothetical stochastic actions – that enable the function approximator to estimate the value of other action probabilities than just those provided by the Actor. This yields the following SARSA-like TD-errors for $\mathbf{p}$, $\mathbf{p}^*$ and $\mathbf{a}$:

$$E^{TD}\langle h_{t-1}, \mathbf{p}_{t-1}\rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{p}_{t-1})$$
$$E^{TD}\langle h_{t-1}, \mathbf{p}_{t-1}^*\rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{p}_{t-1}^*)$$
$$E^{TD}\langle h_{t-1}, \mathbf{a}_{t-1}\rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{a}_{t-1})$$

It seems prudent to choose $\mathcal{P}$ such that $\mathbf{p}^*$ are generated reasonably close to $\mathbf{p}$.

Because of *limited memory*'s inherantly imperfect state-from-history extraction capabilities, there will always be a measure of hidden state present. If the amount of state uncertainty reaches undesirable levels, it may be appropriate *not* to use TD-learning techniques to train the Policy Gradient Critic, since conventional TD-updates are essentially flawed in hidden state situations with discounted payoff [2]. Instead, one would use direct history-to-return mappings. In essence, this can already be accomplished by simply using eligibility traces, which achieves a similar effect as λ approaches 1.

3 Experiments

We carried out experiments on three fundamentally different problem domains. The first task, 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 the agent to learn to remember its initial observation until the end of the episode. The third task, the 89-state Maze [11], is an extremely stochastic discrete control task.

All experiments were carried out with 15-cell LSTMs with direct input-output connections for both Actor and Policy Gradient Critic, and learning took place in batches of 100 sequences (251 in the case of the 89-state Maze). Plain gradient descent and ascent were used for Policy Gradient Critic and Actor, respectively. All experiments used an eligibility trace with $\lambda = 0.8$.

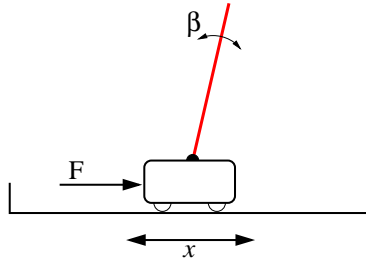### 3.1 Continuous Control: non-Markovian Pole Balancing

Fig. 2. The non-Markov pole balancing task. The task consists of a moving cart on a track, with a pole hinged on top. The controller applies a (continuous) force F to the cart at every time step, after observing the pole angle β (but not the angular velocity, making this a non-Markovian problem). The objective is to indefinitely keep the pole from falling.

This task involves trying to balance a pole hinged to 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 and 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 task more challenging, an extension is made: remove velocity information $\dot{x}$ and $\dot{\beta}$ such that the problem becomes non-Markov. This yields non-Markovian pole balancing [12], a more challenging task.

We applied PGAC to the pole balancing task, using a Gaussian output structure, consisting of a μ output neuron (which was interpreted linearly) and a σ output neuron (which was scaled with the logistic function in order to prevent σ from being negative). Using $\gamma = 0.99$, reward was set to 0.0 at all time steps, except for the last time step when one of the poles falls over, where the reward is -1.0.

A run was considered a *success* when the pole did not fall over for $5,000$ time steps. Averaged over 20 runs, it took $34,823$ evaluations until the success criterion was reached. Interesting is that during learning, often the full stochastic policy had a higher value than the greedy policy (setting σ to 0), showing the usefulness of learning stochastic policies. The results for non-Markovian control clearly outperform most other single-agent memory-based continuous RL methods as far as we are aware (e.g. compare [4]'s finite state controller which cannot hold up the pole for more than 1000 time steps even after half a million evaluations), but some methods that are not single-agent, like evolutionary methods (e.g. [5]), still hold a competitive edge over PGAC.

3.2 Discrete Control: the Long Term Dependency T-maze

The second experiment was carried out on the T-maze [13] (see Figure 3), a discrete control task with output neurons that code for a softmax layer from which an action is
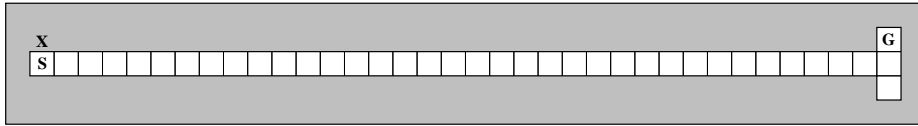
Fig. 3. The T-maze task. The agent observes its immediate surroundings and is capable of the actions north, east, south, and west. It starts in the position labeled 'S', there and only there observing either the signal 'up' or 'down', indicating whether it should go up or down at the T-junction. It receives a reward if it goes in the right direction, and a punishment if not. In this example, the direction is 'up' and N, the length of the alley, is 35.

drawn probabilistically. 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 to the first state's **X**-flag, the agent perceives only its immediate surroundings – whether there is a wall north, east, south or west of it. The agent has four possible actions: North, East, South and West. While in the corridor, if the agent bumps into the wall, it receives a punishment of -0.1, while if it goes east or west, it receives a reward of 0.0. 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 60, and for each length 10 runs were performed. Discount factor $\gamma = 0.98$ was used. In Figure 4 the results are displayed. Using LSTM, PGAC is able to capture the relevant long term dependencies (up to 40 time steps) necessary for solving this task. This is only slightly worse than the best performing algorithm known to solve this task [13] which learns the task up to corridor length 70.

3.3 Discrete Control: the 89-state Maze

In this extremely noisy benchmark task (see Figure 5; see [11] for a complete description) the aim is to get to the goal as fast as possible (where the reward is 1), but within 251 time steps. Discount factor $\gamma = 0.98$ is used.

Good results were achieved for 10 runs of the algorithm. 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 70, and the goal was reached in 85% of cases. This compares favorably with memory-less SARSA(λ) [14], one of the best (and similar) model-free approaches on this task, with numbers 73 steps and 77%, respectively. However, Bakker's RL-LSTM method [15] still clearly outperforms the PGAC algorithm with 61 steps and 93.9%, respectively.
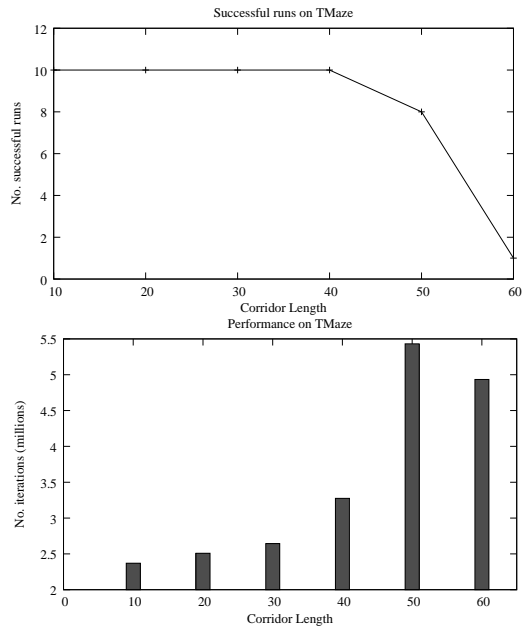
Fig. 4. T-maze results. The upper chart shows the number of successful runs for $N = 10, \ldots, 60$. PGAC Reinforcement Learning's performance starts to degrade at length $N = 50$. The lower plot shows the number of average iterations required to solve the task, averaged over the successful runs.
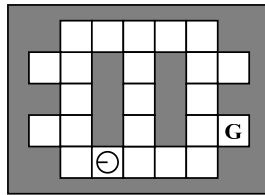

Fig. 5. The 89-state maze. In this extremely stochastic maze, the agent has a position, an orientation, and can execute five different actions: forward, turnleft, turnright, turnabout, and doNothing. The agent starts every trial in a random position. Its goal is to move to the square labeled 'G'. Observations 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 [11] for a complete description of this problem domain. It is interesting to note that, to the authors' knowledge, this domain has as of yet not been satisfactorily solved, that is, solved up to human-comparable performance. That is what makes this a very interesting task.

4   Discussion

Initial results with PGAC Reinforcement Learning show that it is competitive with some of the best approaches on very different benchmark tasks. It does not yet outperform the best available approaches, though. This might be due to two reasons. First, the selection of the perturbation operator has a large influence on estimation variance. Further research into adjusting the choice of this operator might include investigating the appropriate finetuning of perturbations given the entropy in action distributions. Second, since the algorithm uses a limited-memory algorithm, some measure of hidden state remains present. This means that using on-policy temporal difference value updates as discussed above is essentially flawed, although this problem is largely overcome by the use of eligibility traces. A more correct but likely slower approach would involve estimating returns directly from histories.

Since the algorithm addresses learning limited-memory stochastic policies – an under-researched general class of problems which is essential to numerous real-world reinforcement learning tasks – in a simple, natural framework, we hope its performance will be boosted in future research by further analysis and the use of more advanced techniques in, for example, gradient-based learning methods, temporal difference algorithms and DRFA architectures. One area for improvement could include the development of a more principled method for creating action distribution perturbations, or, alternatively, the use of noise in the executed actions while weighting the obtained data points proportionally to their respective probability densities.

Although policy gradient methods can also learn stochastic policies, PGAC is specifically designed to both learn memory and to assign explicit value to stochasticity, making it ideally suited to learning limited-memory stochastic policies. A key feature of the algorithm is that the resulting stochastic policies are not learnt from brute force sampling, but by using an actual Policy Gradient Critic model, with the advantage of generalization and possibly lower estimation variance.

PGAC can be seen as an instance of generalized policy iteration, where value and policy iteratively improve, reinforcing each other. Since a gradient is used to update the action probabilities, it is not guaranteed to converge to a global optimum. However, the use of stochasticity in continuous action spaces holds the promise of overcoming at least part of the sensitivity normally associated with gradient-based continuous reinforcement learning.

5   Conclusion

We have introduced PGAC Reinforcement Learning, a new RL method for learning limited-memory stochastic policies which updates continuous and stochastic policies. A Policy Gradient Critic explicitly attributes value to stochasticity, yielding a flexible algorithm that does not need a prewired exploration strategy since it *learns* to adapt its stochastic action probabilities through experience. Using an appropriate recurrent function approximator, Long Short-Term Memory, the algorithm is capable of solving difficult tasks in environments with long time dependencies and continuous action spaces. Showing competitive results on three benchmark tasks, this algorithm seems promising for extensions to real-world RL tasks.

Acknowledgments

References

1. Sutton, R., Barto, A.: Reinforcement learning: An introduction. Cambridge, MA, MIT Press (1998)
2. Singh, S., Jaakkola, T., Jordan, M.: Learning without state-estimation in partially observable markovian decision processes. In: International Conference on Machine Learning. (1994) 284–292
3. Aberdeen, D.: Policy-Gradient Algorithms for Partially Observable Markov Decision Processes. PhD thesis, Australian National University (2003)
4. Meuleau, N., L., Kim, K., 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
5. Gomez, F.J., Schmidhuber, J.: Co-evolving recurrent neurons learn deep memory POMDPs. In: Proc. of the 2005 conference on genetic and evolutionary computation (GECCO), Washington, D. C., ACM Press, New York, NY, USA (2005)
6. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8) (1997) 1735–1780
7. Werbos, P.: Back propagation through time: What it does and how to do it. In: Proceedings of the IEEE. Volume 78. (1990) 1550–1560
8. Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent networks. Neural Computation **1**(2) (1989) 270–280
9. 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)
10. Schmidhuber, J.: RNN overview, with links to a dozen journal publications (2004) http://www.idsia.ch/˜juergen/rnn.html.
11. Littman, M., Cassandra, A., Kaelbling, L.: Learning policies for partially observable environments: Scaling up. In Prieditis, A., Russell, S., eds.: Machine Learning: Proceedings of the Twelfth International Conference, Morgan Kaufmann Publishers, San Francisco, CA (1995) 362–370
12. 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
13. Bakker, B.: Reinforcement learning with long short-term memory. In: Advances in Neural Information Processing Syst., 14, (2002). (2002)
14. Loch, J., Singh, S.: Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In: Proc. 15th International Conf. on Machine Learning, Morgan Kaufmann, San Francisco, CA (1998) 323–331
15. Bakker, B.: The State of Mind: : Reinforcement Learning with Recurrent Neural Networks. PhD thesis, Leiden University (2004)