Policy Gradients with Parameter-Based Exploration for Control

Frank Sehnke[1], Christian Osendorfer[1], Thomas Rückstieß[1], Alex Graves[1], Jan Peters[3], and Jürgen Schmidhuber[1,2]

[1] Faculty of Computer Science, Technische Universität München, Germany
[2] IDSIA, Manno-Lugano, Switzerland
[3] Max-Planck Institute for Biological Cybernetics Tübingen, Germany

Abstract. We present a model-free reinforcement learning method for partially observable Markov decision problems. Our method estimates a likelihood gradient by sampling directly in parameter space, which leads to lower variance gradient estimates than those obtained by policy gradient methods such as REINFORCE. For several complex control tasks, including robust standing with a humanoid robot, we show that our method outperforms well-known algorithms from the fields of policy gradients, finite difference methods and population based heuristics. We also provide a detailed analysis of the differences between our method and the other algorithms.

1 Introduction

Policy gradient methods, so called because they search in policy space without using value estimation, are among the most effective optimisation strategies for complex, high dimensional reinforcement learning tasks [1,2,3,4]. However, a significant problem with policy gradient algorithms such as REINFORCE [5], is that the high variance in their gradient estimates leads to slow convergence. Various approaches have been proposed to reduce this variance [6,7,2,8].

In what follows we introduce an alternative method, called policy gradients with parameter-based exploration (PGPE), which replaces the search in policy space with a direct search in model parameter space. As with REINFORCE, the search is carried out by generating history samples, and using these to estimate the likelihood gradient with respect to the parameters. The advantage of PGPE is that a single parameter sample can be used to generate an entire action-state history, in contrast with policy gradient methods, where an action sample is drawn from the policy on every time step. This provides PGPE with lower variance history samples, and correspondingly lower variance gradient estimates. In addition, since PGPE estimates the parameter gradient directly, it can be used to train non-differentiable controllers.

The PGPE algorithm is derived in detail in Section 2. In Section 3, we test PGPE on three control experiments, and compare its performance with REINFORCE, evolution strategies (ES) [9], simultaneous perturbation stochastic adaptation (SPSA) [10], and natural actor critic (NAC) [4]. In Section 4 we

V. Kurkova, R. Neruda, and J. Koutnik (Eds.): ICANN 2008, Part I, LNCS 5163, pp. 387–396, 2008.

analyse the relationship between PGPE and the other algorithms. In particular we carry out a range of experiments where we iteratively modify each of REINFORCE, SPSA and ES in such a way that they become more like PGPE, and evaluate the corresponding improvement in performance. Conclusions and directions for future work are presented in Section 5.

2 The Policy Gradients with Parameter-Based Exploration Method

In what follows we derive the PGPE algorithm from the general framework of episodic reinforcement learning in a Markovian environment. In doing so we highlight the differences between PGPE and policy gradient methods such as REINFORCE, and discuss why these differences lead to more accurate parameter gradient estimates.

Consider an agent interacting with an environment. Denote the state of environment at time t as s_t and the action at time t as a_t. Because we are interested in continuous state and action spaces (usually required for control tasks), we represent both a_t and s_t with real valued vectors. We assume that the environment is Markovian, i.e. that the current state-action pair defines a probability distribution over the possible next states $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$. We further assume that the actions depend stochastically on the current state and some real valued vector θ of agent parameters: $a_t \sim p(a_t|s_t, \theta)$. Lastly, we assume that each state-action pair produces a scalar reward $r_t(a_t, s_t)$. We refer to a length T sequence of state-action pairs produced by an agent as a *history* $h = [s_{1:T}, a_{1:T}]$ (elsewhere in the literature such sequences are referred to as *trajectories* or *roll-outs*).

Given the above formulation we can associate a cumulative reward r with each history h by summing over the rewards at each time step: $r(h) = \sum_{t=1}^{T} r_t$. In this setting, the goal of reinforcement learning is to find the parameters θ that maximize the agent's expected reward:

$$J(\theta) = \int_H p(h|\theta) r(h) dh \tag{1}$$

An obvious way to maximise J is to use $\nabla_\theta J$ to carry out gradient ascent. Noting that the reward for a particular history is independent of θ, and using the standard identity $\nabla_x y(x) = y(x) \nabla_x \log x$, we can write

$$\nabla_\theta J(\theta) = \int_H p(h|\theta) \nabla_\theta \log p(h|\theta) r(h) dh \tag{2}$$

Since the environment is Markovian, and since the states are conditionally independent of the parameters given the agent's choice of actions, we can write $p(h|\theta) = p(s_1) \Pi_{t=1}^{T} p(s_{t+1}|s_t, a_t) p(a_t|s_t, \theta)$. Substituting this into Eq. (2) gives

$$\nabla_\theta J(\theta) = \int_H p(h|\theta) \sum_{t=1}^{T} \nabla_\theta p(a_t|s_t, \theta) r(h) dh \tag{3}$$

Clearly, integrating over the entire space of histories is unfeasible, and we therefore resort to sampling methods:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta p(a_t^i | s_t^i, \theta) r(h^i) \tag{4}$$

where the histories h^i are chosen according to $p(h^i|\theta)$. The question then becomes one of how to model $p(a_t|s_t, \theta)$. In policy gradient methods such as REINFORCE, the parameters θ are used to determine a probabilistic *policy* $\pi_\theta(a_t|s_t) = p(a_t|s_t, \theta)$. A typical policy model would be a parametric function approximator whose outputs define the probabilities of taking different actions. In this case the histories can be sampled by choosing an action at each time step according to the policy distribution, and the final gradient can be calculated by differentiating the policy with respect to the parameters. However, the problem is that sampling from the policy on every time step leads to an excessively high variance in the sample over histories, and therefore in the estimated gradient.

PGPE addresses the variance problem by replacing the probabilistic policy with a probability distribution over the parameters themselves, that is

$$p(a_t|s_t, \rho) = \int_\Theta p(\theta|\rho) \delta_{F_\theta(s_t), a_t} d\theta, \tag{5}$$

where ρ are the hyperparameters determining the distribution over the parameters θ, $F_\theta(s_t)$ is the (deterministic) action chosen by the model with parameters θ in state s_t, and δ is the usual Dirac delta function. The advantage of this approach is that, because the actions are deterministic, an entire history can be generated using a single sample from the parameters, thereby reducing the variance in the gradient estimate. As an added benefit the gradient is estimated directly by sampling the parameters, which allows the use of non-differentiable controllers. The expected reward with hyperparameters ρ is:

$$J(\rho) = \int_\Theta \int_H p(h, \theta|\rho) r(h) dh d\theta \tag{6}$$

Differentiating this with respect to ρ and applying the log trick as before we get:

$$\nabla_\rho J(\rho) = \int_\Theta \int_H p(h, \theta|\rho) \nabla_\rho \log p(h, \theta|\rho) r(h) dh d\theta \tag{7}$$

Noting that h is conditionally independent of ρ given θ, we have $p(h, \theta|\rho) = p(h|\theta)p(\theta|\rho)$ and therefore $\nabla_\rho \log p(h, \theta|\rho) = \nabla_\rho p(\theta|\rho)$. Substituting this into Eq. (6) we get

$$\nabla_\rho J(\rho) = \int_\Theta \int_H p(h|\theta) p(\theta|\rho) \nabla_\rho \log p(\theta|\rho) r(h) dh d\theta \tag{8}$$

Again we approximate the above by sampling, this time by first choosing θ from $p(\theta|\rho)$, then running the agent to generate h from $p(h|\theta)$. In what follows we

assume that ρ consists of a set of means μ_i and standard deviations σ_i that determine an independent normal distribution for each parameter θ_i in θ . Note that more complex forms for the dependency of θ on ρ could be used, at the expense of higher computational cost. Some rearrangement gives the following forms for the derivative of $\log p(\theta|\rho)$ with respect to μ_i and σ_i:

$$\nabla_{\mu_i} \log p(\theta|\rho) = \frac{(\theta_i - \mu_i)}{\sigma_i^2} \qquad \nabla_{\sigma_i} \log p(\theta) = \frac{(\theta_i - \mu_i)^2 - \sigma_i^2}{\sigma_i^3} \tag{9}$$

Following Williams [5], we update each σ_i and μ_i in the direction of the gradient using a step size $\alpha_i = \alpha\sigma_i^2$, where α is a constant. Also following Williams we subtract a baseline b from the reward r for each history. This gives us the following hyperparameter update rules:

$$\Delta\mu_i = \alpha(r-b)(\theta_i - \mu_i) \qquad \Delta\sigma_i = \alpha(r-b)\frac{(\theta_i - \mu_i)^2 - \sigma_i^2}{\sigma_i} \tag{10}$$

A possible objection to PGPE is that parameter space is generally higher dimensional than action space, and therefore has higher sampling complexity. However, recent results [11] indicate that this drawback was overestimated in the past. In this paper we present experiments where PGPE successfully trains a controller with more than 1000 parameters. Another issue is that PGPE, at least in its present form, is explicitly episodic, since the parameter sampling is carried out once per history. This contrasts with policy gradient methods, which can be applied to infinite horizon settings as long as frequent rewards can be computed.

3 Experiments

In this section we compare PGPE with SPSA, REINFORCE, NAC and ES, on three simulated control scenarios. These scenarios allow us to model problems of similar complexity to today's real-life RL problems [12,2].

In all experiments involving evolution strategies (ES) we used a local mutation operator. We did not examine correlated mutation and CMA-ES because both mutation operators add $n(n-1)$ strategy parameters to the genome. Since we have more than 1000 parameters for the largest controller, this would lead to a prohibitive memory cost. In addition, the local mutation operator is more similar to the perturbations in PGPE, making it easier to compare the algorithms. All plots show the average results of 40 independent runs.

3.1 Pole Balancing

The first scenario is the standard pole balancing benchmark [11]. In this task the agent's goal is to maximize the length of time a movable cart can balance a pole upright in the center of a track. The agent's inputs are the angle and angular velocity of the pole and the position and velocity of the cart. The agent is represented by a linear controller with four inputs and one output. The simulation is

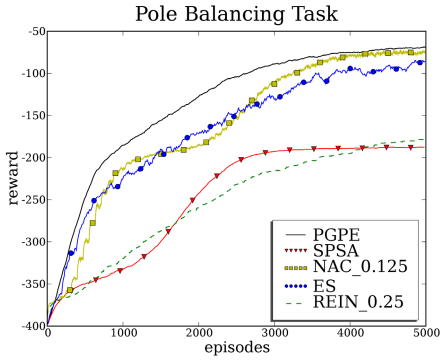

Fig. 1. PGPE compared to ES, SPSA, REINFORCE and NAC on the pole balancing benchmark

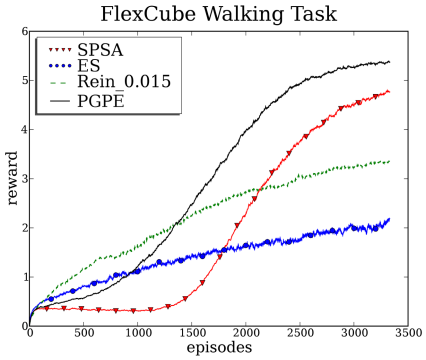

Fig. 2. PGPE compared to ES, SPSA and REINFORCE on the walking task

updated 50 times a second. The initial position of the cart and angle of the pole are chosen randomly. Figure 1 shows the performance of the various methods on the pole balancing task. All algorithms quickly learned to balance the pole, and all but SPSA eventually learned to do so in the center of the track. PGPE was both the fastest to learn and the most effective algorithm on this benchmark.

3.2 FlexCube Walking Task

The second scenario was a mass-particle system with 8 particles. The particles are modelled as point masses on the vertices of a cube, with every particle connected to every other by a spring (see Fig. 3). We refer to this scenario as the *FlexCube* framework. Though relatively simple, FlexCube can be used to perform sophisticated tasks with continuous state and action spaces. In this case the task is to make the cube "walk" — that is, to maximize the distance of its center of gravity from the starting point.

The agent can control the desired lengths of the 12 edge springs. Its inputs are the 12 current edge spring lengths, the 12 previous desired edge spring lengths (fed back from its own output at the last time step) and the 8 floor contact sensors in the vertices. The agent is represented by a Jordan network [13] with 32 inputs, 10 hidden units and 12 output units. Figure 2 shows the results on the walking task. All the algorithms learn to move the FlexCube. However, for reasons that are unclear to the authors, ES is only able to do so very slowly. PGPE substantially outperforms the other methods, both in learning speed and final reward. For a detailed view of the solutions in the walking task please refer to the video on http://www.pybrain.org/videos/icann08/.

3.3 Biped Robot Standing Task

The task in this scenario was to keep a simulated biped robot standing while perturbed by external forces. The simulation, based on the biped robot

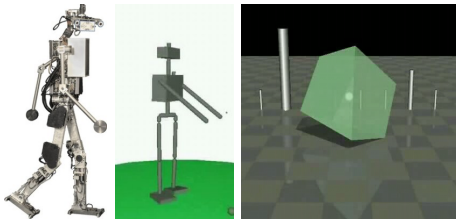

Fig. 3. The real Johnnie robot (left), its simulation (center) and the FlexCube (right)

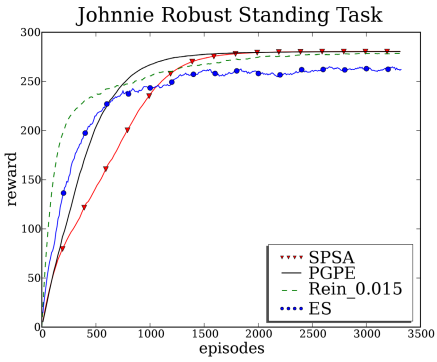

Fig. 4. PGPE compared to ES, SPSA and REINFORCE on the robust standing benchmark

Johnnie [14] was implemented using the Open Dynamics Engine. The lengths and masses of the body parts, the location of the connection points, and the range of allowed angles and torques in the joints were matched with those of the original robot. Due to the difficulty of accurately simulating the robot's feet, the friction between them and the ground was approximated with Coulomb friction. The framework has 11 degrees of freedom and a 41 dimensional observation vector (11 angles, 11 angular velocities, 11 forces, 2 pressure sensors in feet, 3 degrees of orientation and 3 degrees of acceleration in the head). The controller was a Jordan network [13] with 41 inputs, 20 hidden units and 11 output units.

The aim of the task is to maximize the height of the robot's head, up to the limit of standing completely upright. The robot is continually perturbed by random forces (see Figure 5) that would knock it over if it did not react.

As can be seen from the results in Fig. 4, the task was relatively easy, and all the methods were able to quickly achieve a high reward. REINFORCE learned especially quickly, and outperformed PGPE in the early stages of learning. However PGPE overtook it after about 500 training episodes. Figure 5 shows a typical scenario of the robust standing task with a reward outcome of 279. For more detailed views of the solution please refer to the video on http://www.pybrain.org/videos/icann08/.

3.4 Discussion

One general observation from our experiments was that the longer the episodes, the more PGPE outperformed policy gradient methods. This is not surprising, since the variance of the REINFORCE gradient estimates increases with the number of action samples. However it is an important benefit, given that most interesting real-world problems require much longer episodes than our experiments [1,2,12].

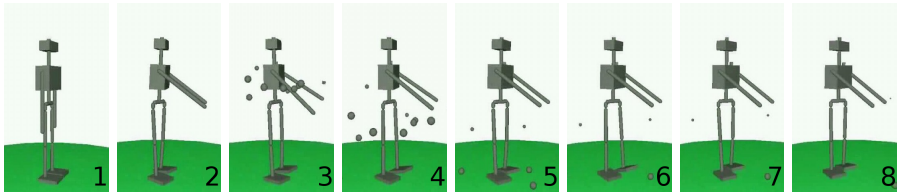

Fig. 5. From left to right, a typical solution which worked well in the robust standing task is shown: 1. Initial posture. 2. Stable posture. 3. Perturbation. 4. - 7. Backsteps right, left, right, left. 8. Stable posture regained.

4 Relationship to Other Algorithms

In this section we quantify the differences between PGPE and SPSA, ES and REINFORCE, and assess the impact of these differences on performance. Starting with each of other algorithms we incrementally alter them so that their behaviour (and performance) becomes closer to that of PGPE. In the case of SPSA we end up an algorithm identical to PGPE; for the other methods the transformed algorithm is similar, but still inferior, to PGPE.

4.1 From SPSA to PGPE

Two changes are required to transform SPSA into PGPE. First the uniform sampling of perturbations is replaced by Gaussian sampling, (with the finite differences gradient correspondingly replaced by the likelihood gradient). Second, the variances of the perturbations are turned into free parameters and trained with the rest of the model (initially the Gaussian sampling is carried out with fixed variance, just as the range of uniform sampling is fixed in SPSA). Figure 6 shows the performance of the three variants of SPSA on the walking task. Note that the final variant is identical to PGPE (solid line). For this task the main improvement comes from the switch to Gaussian sampling and the likelihood gradient (circles). Adding adaptive variances actually gives slightly slower learning at first, although the two converge later on. The original parameter update rule for SPSA is:

$$\theta_i(t+1) = \theta_i(t) - \alpha \frac{y_+ - y_-}{2\epsilon} \tag{11}$$

with $y_+ = r(\theta + \Delta\theta)$ and $y_- = r(\theta - \Delta\theta)$, where $r(\theta)$ is the evaluation function and $\Delta\theta$ is drawn from a Bernoulli distribution scaled by the time dependent step size $\epsilon(t)$, i.e. $\Delta\theta_i(t) = \epsilon(t) \cdot rand[-1, 1]$ In addition, a set of metaparameters is used to help SPSA converge. ϵ decays according to $\epsilon(t) = \frac{\epsilon(0)}{t^\gamma}$ with $0 < \gamma < 1$. Similarly, α decreases over time, with $\alpha = a/(t + A)^E$ for some fixed a, A and E [10]. The choice of initial parameters $\epsilon(0), \gamma$, a, A and E is critical to the performance of SPSA. To switch from uniform to Gaussian sampling we simply modify the perturbation function to $\Delta\theta_i(t) = \mathcal{N}(0, \epsilon(t))$. We then follow the derivation of the likelihood gradient outlined in Section 2, to obtain the same

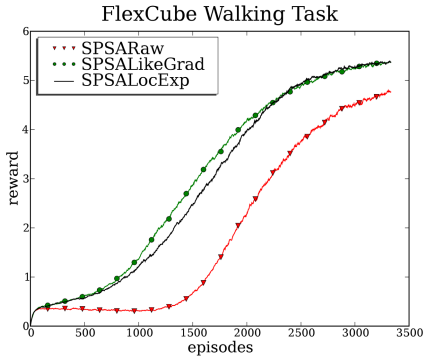

Fig. 6. Three variants of SPSA on the FlexCube walking task: the original algorithm (SPSARaw), the algorithm with normally distributed sampling and likelihood gradient (SPSALikeGrad), and with adaptive variance (SPSALocExp)

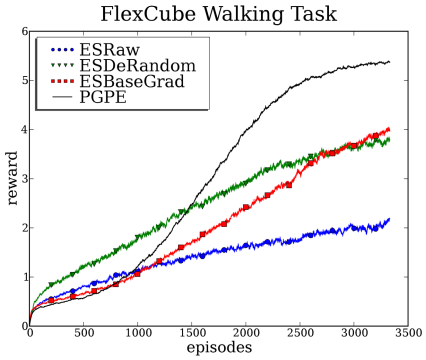

Fig. 7. Three variants of ES on the FlexCube walking task: the original algorithm (ESRaw), derandomized ES (ESDeRandom) and gradient following (ESBaseGrad). PGPE is shown for reference.

parameter update rule as used for PGPE (Eq. (10)). The correspondence with PGPE becomes exact when we calculate the gradient of the expected reward with respect to the sampling variance, giving us the standard deviation update rule stated in Eq. (10). As well as improved performance, the above modifications greatly reduce the number of free parameters in the algorithm. The SPSA metaparameters are now condensed to: a step size α_μ for updating the parameters, a step size α_σ for updating the standard deviations of the perturbations, and an initial value standard deviation σ_{init}. Furthermore, we found that the parameters $\alpha_\mu = 0.2$, $\alpha_\sigma = 0.1$ and $\sigma_{init} = 2.0$ worked very well for a wide variety of tasks.

4.2 From ES to PGPE

We now examine the effect of two modifications that largely bridge the gap between ES and PGPE. First we switch from standard ES to derandomized ES [15], which somewhat resembles the gradient based variance updates found in PGPE. We then change from using population based search to following a likelihood gradient. The results are plotted in Figure 7. As can be seen, both modifications bring significant improvements, although neither can match PGPE. While ES performs well initially, it is slow to converge to good optima. This is partly because, as well as having stochastic mutations, ES has stochastic updates to the standard deviations of the mutations, and the coupling of these two stochastic processes slows down convergence. Derandomized ES alleviates that problem by using instead a deterministic standard deviation update rule, based on the change in parameters between the parent and child. Tracking a population has advantages in the early phase of search, when broad, relatively undirected exploration is desirable. This is particularly true for the multimodal fitness spaces

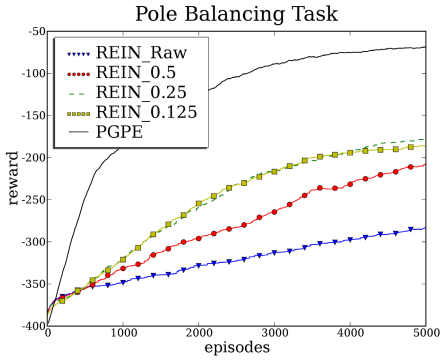

Fig. 8. REINFORCE on the pole balancing task, with various frequencies of action perturbation. PGPE is shown for reference.

typical of realistic control tasks. However in later phases convergence is usually more efficient with gradient based methods. Applying the likelihood gradient, the ES parameter update rule becomes:

$$\Delta\theta_i = \alpha \sum_{m=1}^{M} (r_m - b)(y_{m,i} - \theta_i), \tag{12}$$

where M is the number of samples and $y_{m,i}$ is parameter i of sample m.

4.3 From REINFORCE to PGPE

We previously asserted that the lower variance of PGPE's gradient estimates is partly due to the fact that PGPE requires only one parameter sample per history, whereas REINFORCE requires samples every time step. This suggests that reducing the frequency of REINFORCE perturbations should improve its gradient estimates, thereby bringing it closer to PGPE. Fig. 8 shows the performance of episodic REINFORCE with a perturbation probability of 1, 0.5, 0.25, and 0.125 per time step. In general, performance improved with decreasing perturbation probability. However the difference between 0.25 and 0.125 is negligible. This is because reducing the number of perturbations constrains the range of exploration at the same time as it reduces the variance of the gradient, leading to a saturation point beyond which performance does not increase. Note that the above trade off does not exist for PGPE, because a single perturbation of the parameters can lead to a large change in behaviour.

5 Conclusion and Future Work

We have introduced PGPE, a novel algorithm for episodic reinforcement learning based on a gradient based search through model parameter space. We derived the PGPE equations from the basic principle of reward maximization, and

explained why they lead to lower variance gradient estimates than those obtained by policy gradient methods. We compared PGPE to a range of stochastic optimisation algorithms on three control tasks, and found that it gave superior performance in every case. Lastly we provided a detailed analysis of the relationship between PGPE and the other algorithms. One direction for future work would be to establish whether Williams' local convergence proofs for REINFORCE can be generalised to PGPE. Another would be to combine PGPE with recent improvements in policy gradient methods, such as natural gradients and base-line approximation [4].

Acknowledgement. This work is supported within the DFG excellence research cluster "Cognition for Technical Systems - CoTeSys", www.cotesys.org

References

1. Benbrahim, H., Franklin, J.: Biped dynamic walking using reinforcement learning. Robotics and Autonomous Systems Journal (1997)
2. Peters, J., Schaal, S.: Policy gradient methods for robotics. In: IROS-2006, Beijing, China, pp. 2219–2225 (2006)
3. 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, vol. 18. MIT Press, Cambridge (2006)
4. Peters, J., Vijayakumar, S., Schaal, S.: Natural actor-critic. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 280–291. Springer, Heidelberg (2005)
5. Williams, R.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning 8, 229–256 (1992)
6. Baxter, J., Bartlett, P.L.: Reinforcement learning in POMDPs via direct gradient ascent. In: Proc. 17th International Conf. on Machine Learning, pp. 41–48. Morgan Kaufmann, San Francisco (2000)
7. Aberdeen, D.: Policy-Gradient Algorithms for Partially Observable Markov Decision Processes. PhD thesis, Australian National University (2003)
8. Sutton, R., McAllester, D., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: NIPS 1999, pp. 1057–1063 (2000)
9. Schwefel, H.: Evolution and optimum seeking. Wiley, New York (1995)
10. Spall, J.: An overview of the simultaneous perturbation method for efficient optimization. Johns Hopkins APL Technical Digest 19(4), 482–492 (1998)
11. Riedmiller, M., Peters, J., Schaal, S.: Evaluation of policy gradient methods and variants on the cart-pole benchmark. In: ADPRL 2007 (2007)
12. Müller, H., Lauer, M., Hafner, R., Lange, S., Merke, A., Riedmiller, M.: Making a robot learn to play soccer. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNCS (LNAI), vol. 4667, pp. 220–234. Springer, Heidelberg (2007)
13. Jordan, M.: Attractor dynamics and parallelism in a connectionist sequential machine. In: Proc. of the Eighth Annual Conference of the Cognitive Science Society, vol. 8, pp. 531–546 (1986)
14. Ulbrich, H.: Institute of Applied Mechanics, TU München, Germany (2008), `http://www.amm.mw.tum.de/`
15. Hansen, N., Ostermeier, A.: Completely Derandomized Self-Adaptation in Evolution Strategies. Evolutionary Computation 9(2), 159–195 (2001)