Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)

Evaluation of Policy Gradient Methods and Variants on the Cart-Pole Benchmark

Martin Riedmiller
Neuroinformatics Group
University of Osnabrueck
Email: martin.riedmiller@uos.de

Jan Peters, Stefan Schaal
Computational Learning and Motor Control
University of Southern California
Email: {jrpeters,sschaal}@usc.edu

***Abstract*— In this paper, we evaluate different versions from the three main kinds of model-free policy gradient methods, i.e., finite difference gradients, 'vanilla' policy gradients and natural policy gradients. Each of these methods is first presented in its simple form and subsequently refined and optimized. By carrying out numerous experiments on the cart pole regulator benchmark we aim to provide a useful baseline for future research on parameterized policy search algorithms. Portable C++ code is provided for both plant and algorithms; thus, the results in this paper can be reevaluated, reused and new algorithms can be inserted with ease.**

I. Introduction

Recently, there has been a strong push in the reinforcement learning community towards creating solid standards which allow the evaluation and comparison for different reinforcement learning methods [12]. In this spirit, with the work presented in this paper, we intend to take an area of reinforcement learning, i.e., model-free policy gradient methods and attempt to create useful baselines by providing benchmarking simulation setups and well-refined algorithms.

We compare the most important different policy gradients algorithms employing a variety of different versions of each algorithm. The results for these algorithms should serve as a starting point for the evaluation of future parameterized policy search algorithms. As a considerable effort was put into design and implementation of each algorithm as well as the learning meta parameters (e.g., learning rates, exploration initialization, etc). Thus, we hope to provide a solid baseline for researchers advocating their own algorithms, variants, heuristics or parameter choices. The software and documentation for all algorithms and evaluated plants is provided online (www.ni.uos.de/pgmethods).

The second goal of this paper is to highlight some of the most important properties which need to be dealt with when applying policy gradient methods. This aim requires that the chosen plant is well-known, easily accessible, neither too difficult nor too easy to control. The later requirement is particularly important as highly complex plants, e.g., anthromorphic robot arms [4], [8] or legged robots [2], [6], [13], require highly task specific adaptation of both the policy and the algorithm. Thus, in order to ensure the possibility of generalizing to different domains, we are focussing here on a standard problem, i.e., cart-pole regulation in a deterministic and stochastic version.

In the remainder of this paper, we will proceed as follows. First, we will review the different basic algorithms for estimating policy gradients [8], and, additionally, how to combine them with a step-size adaptation method known from supervised learning, i.e., Rprop [10]. Second, we will present the setup in order to compare policy gradient reinforcement learning methods, and, finally, show results and a conclusion.

II. Policy Gradient Methods: Algorithms & Variants

Policy gradient methods, similar to most other reinforcement learning methods, consist of two steps, i.e., (i) a policy evaluation step which results into an estimate of the gradient $\nabla_\theta J(\theta)$ of the expected return

$$J(\theta) = E\left\{ \sum\nolimits_{t=0}^{N} r_t \middle| \pi_\theta \right\} \tag{1}$$

for the current policy π_θ (where r_t denotes the reward at time t) and (ii) a policy improvement step which is realized by updating the policy parameters through steepest gradient ascent

$$\theta_{k+1} = \theta_k + \alpha_k \nabla_\theta J(\theta) \tag{2}$$

where α_k denotes a learning rate.

For policy gradient estimation, we consider three different approaches, i.e., finite difference methods (FD) which perturb the policy parameters of a deterministic policy, vanilla policy gradient methods (VPG) and natural policy gradient methods (NG) which both perturb the motor commands by employing a stochastic policy [8]. The three different methods are described in the following sections. However, we will not give an in-depth account of each method but rather present the basic behind each of them and refer to the literature for detailed derivations.

For computing the policy update, we compare standard gradient descent updates with constant learning rates $\alpha_k = \alpha$ with an update rules based on the Rprop method [10]. The Rprop method is a variable step-size method based on the sign of the partial derivatives rather than on the magnitude. Thus, it is usually significantly more robust and does not require significant manual tuning.

A. Finite Difference (FD) Gradient Estimators

The basic idea behind finite difference methods is that when we intend to estimate gradients from roll-outs, a very simple way is to change the current policy parameters θ_k by small perturbations $\delta\theta_i$ and thus allowing the regression of parameter differences $\delta\theta_i$ onto the resulting performance differences $\delta J_i = J(\theta_k + \delta\theta_i) - J(\theta_k)$. This allows the regression of the parameter perturbations $\mathbf{\Delta\Theta} = [\delta\theta_i]$ onto $\mathbf{\Delta}J = [\delta J_i]$ resulting into a gradient estimate of

$$\mathbf{g}_{\text{FD}} = (\mathbf{\Delta\Theta}^T\mathbf{\Delta\Theta})^{-1}\mathbf{\Delta\Theta}^T\mathbf{\Delta}J. \tag{3}$$

In the case, where the policy parameters are perturbed sequentially after each other, the algorithm becomes particularly simple as the matrix $\mathbf{\Delta\Theta}^T\mathbf{\Delta\Theta}$ just becomes a diagonal.

The advantages of the finite-difference approach are that it is straightforward to implement, very little noise is introduced into the system due to exploration, requires no model and it can be used with deterministic policies in the form $\mathbf{u} = \pi(\mathbf{x})$ where $\mathbf{u}$ denotes the action and $\mathbf{x}$ denotes the state. However, the approach also comes with larger handicaps, e.g., each parameter is differently sensitive to parameter perturbations and, thus, the sizes of the required parameter perturbations can differ by orders of magnitude. Furthermore, the gradient estimation is very sensitive to perturbations which produce extremely bad results, e.g., when a policy is destabilized. Finally, in the presence of noise, the gradient estimate can be very noisy; this can only be fought in simulation where fixing the noise history, e.g., by resetting the seed of the random generator, can reduce the variance significantly [3], [7]. The simplest such an algorithm is shown below.

algorithm: Finite Difference Gradient Evaluation	
input: policy θ	
1	Set seed of random generator to fixed value.
2	**for** each parameter θ_i **do**
3	evaluate $J_0 = \sum_{t=0}^{N} r_t$ with $\theta_i^0 = \theta_i$
4	**for** $k = 1$ **to** K **do**
5	draw $\delta\theta_i^k \sim \text{Uniform}(\delta\theta_{\min}, \delta\theta_{\max})$
6	set $\theta_i^k = \theta_i + \delta\theta_i^k$
7	evaluate $J_k = \sum_{t=0}^{N} r_t$ with θ_i^k
8	**end**.
9	Estimate gradient component $g_i = \frac{\sum \delta\theta_i^k (J_k - J_0)}{\sum \left(\delta\theta_i^k\right)^2}$.
10	**end**.
output: gradient estimate $\mathbf{g}$	

B. 'Vanilla' Policy Gradients (VPG)

Alternatively to perturbing the parameters θ_k of a deterministic policy $\mathbf{u} = \pi(\mathbf{x})$, one could choose a stochastic policy $\mathbf{u} \sim \pi(\mathbf{u}|\mathbf{x})$ such that $\mathbf{u} = \mu(\mathbf{x}) + \epsilon$ where ϵ is a perturbation of the nominal motor command $\mu(\mathbf{x})$. In this case, one can make use of the likelihood ratio trick, i.e., if we rewrite Eq.(1) in terms of trajectories τ with probability

$$p(\tau|\theta) = p(\mathbf{x}_0)\prod_{t=0}^{N-1} p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)\pi(\mathbf{u}_t|\mathbf{x}_t) \tag{4}$$

and summed reward $R(\tau) = \sum_{t=0}^{N} r_t$ and differentiate, we obtain

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \int p(\tau|\theta)R(\tau)d\tau = \int \nabla_\theta p(\tau|\theta)R(\tau)d\tau \\ &= \int p(\tau|\theta)\nabla_\theta \log p(\tau|\theta)R(\tau)d\tau \\ &= E\{\nabla_\theta \log p(\tau|\theta)R(\tau)\}\end{aligned} \tag{5}$$

as $\nabla_\theta p(\tau|\theta) = p(\tau|\theta)\nabla_\theta \log p(\tau|\theta)$ by definition. If we additionally make use of Eq.(4), we see that

$$\nabla_\theta \log p(\tau|\theta) = \sum_{t=0}^{N-1} \nabla_\theta \log \pi(\mathbf{u}_t|\mathbf{x}_t) \tag{6}$$

and thus we can compute the gradient from samples without a model of the system.

However, such a gradient estimate while unbiased, its variance depends on the squared average magnitude of the rewards $R(\tau)$ and thus can become quite large. However, there are two important insights which allow the reduction of the variance. First, as the rewards only depend on previous actions, we realize that all terms $\nabla_\theta \log \pi(\mathbf{u}_t|\mathbf{x}_t)r_h$ with $h < t$ can only introduce variance but will always average out in expectation. A second variance reduction technique is given by the *baselines*, i.e., since we know that $\int p(\tau|\theta)d\tau = 1$, we also have

$$\int p(\tau|\theta)\nabla_\theta \log p(\tau|\theta)d\tau = \nabla_\theta \int p(\tau|\theta)d\tau = \nabla_\theta 1 = 0$$

and thus the gradient estimator can be rewritten as $\nabla_\theta J(\theta) = E\{\nabla_\theta \log p(\tau|\theta)(R(\tau) - b)\}$ where b is a constant. The baseline can be selected to minimize the variance of the gradient estimate which then results into a minimum variance unbiased policy gradient gradient estimator, see [8] for details and the description below for a highly optimized implementation with optimal baselines.

algorithm:'Vanilla' Policy Gradient Evaluation	
input: policy θ, paths $\tau_1, \ldots, \tau_s$, rewards $r_t(\tau)$	
1	**for** each parameter θ_i **do**
2	compute optimal baseline $b_t^i = \frac{\sum_{p=1}^{s}\left(\sum_{h=0}^{t} \nabla_{\theta_i} \log \pi(\mathbf{u}_h^p|\mathbf{x}_h^p)\right)^2 r_t}{\sum_{p=1}^{s}\left(\sum_{h=0}^{t} \nabla_{\theta_i} \log \pi(\mathbf{u}_h^p|\mathbf{x}_h^p)\right)^2}$
3	compute gradient component $g_i = \frac{1}{s}\sum_{p=1}^{s}\sum_{t=0}^{N-1}\sum_{h=0}^{t} \nabla_{\theta_i} \log \pi(\mathbf{u}_h^p|\mathbf{x}_h^p)\left(r_t - b_t^i\right)$
4	**end**.
output: gradient estimate $\mathbf{g}$	

C. Natural Policy Gradients (NPG)

One of the disadvantages of 'vanilla' policy gradients is that even with an optimal baseline, their convergence to the optimal solution can be rather slow. The reason for this is that they do not take into account how much information is lost during the policy update and thus often tend to minimize the exploration instead of exploiting the knowledge obtained in the gradient estimation as illustrated in Figure 1 (a). One principled approach to alleviate this problem is the natural gradient approach [1], [5] which has resulted into several

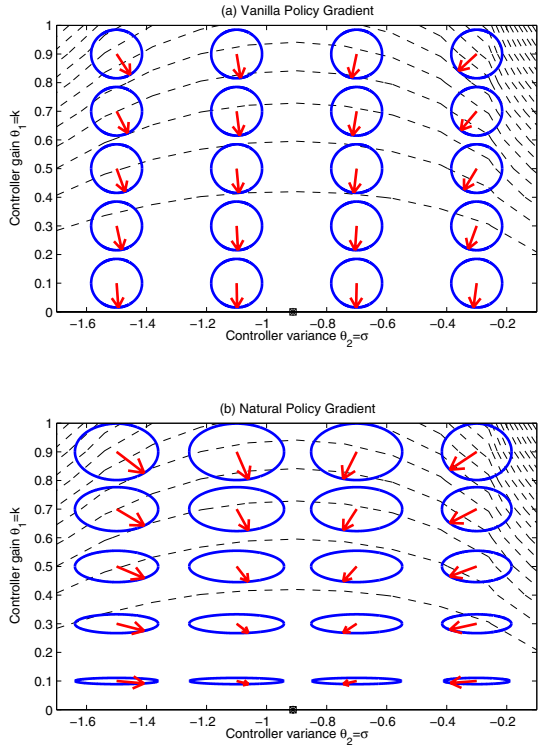

Fig. 1. The classical example of LQR can be used to illustrate why 'vanilla' policy gradients reduce the exploration to zero while natural policy gradients go for the optimal solution.

fast policy gradient learning algorithms such as the Natural Actor-Critic algorithms [9]. The basic idea behind this class of algorithms is that the information about the policy parameters θ contained in the observed paths τ is given by the Fisher information $F(\theta)$ defined as

$$F(\theta) = E\{\nabla_\theta \log p(\tau|\theta) \nabla_\theta \log p(\tau|\theta)^T\}, \qquad (7)$$

i.e., it is the variance of the path derivatives $\nabla_\theta \log p(\tau|\theta)$. If we change the policy by a $\delta\theta$, we occur an information loss $l_\theta(\delta\theta) \approx \delta\theta^T F(\theta)\delta\theta$ which can also be seen as the size of the change in path distribution $p(\tau|\theta)$. Thus, if we search for the policy change $\delta\theta$ which maximizes the expected return $J(\theta + \delta\theta)$ for a constant information loss $l_\theta(\delta\theta) \approx \epsilon$, we basically search for the highest value on an ellipse around θ and go for the direction of this highest value. This direction is illustrated in Figure 1 (b). By contrast, 'vanilla' policy gradients do not take the information loss into account and in policy into account and, thus, correspond to searching for the highest value on a circle around θ as demonstrated in Figure 1 (a). More formalized, we realize that the direction of steepest ascent on the ellipse corresponds to

$$\delta\theta = \text{argmax}_{\delta\theta \text{ s.t. } l(\delta\theta)=\varepsilon} \delta\theta^T \nabla_\theta J = F^{-1}(\theta) \nabla_\theta J. \qquad (8)$$

On the first inspection, this seems to result into more difficult algorithms; however, there is a class of straightforward algorithms which can estimate natural policy gradients using an appropriate baseline, i.e., the Natural Actor-Critic algorithms [9]. In the episodic form, the simplest instantiation of these algorithms is very simple to implement and given below. The derivation of this algorithm is beyond the scope of the paper (for more information on it see [8], [9]) but it corresponds to an unbiased natural gradient estimator with an optimal baseline.

algorithm: Episodic Natural Actor-Critic (eNAC)

input: policy θ, paths $\tau_1, \ldots, \tau_s$, rewards $r_t^1, \ldots, r_t^p$ with $t \in \{1, \ldots, N-1\}$

Compute sufficient statistics

1 $$\mathbf{X} = \begin{bmatrix} \sum_{t=0}^{N-1} \nabla_\theta \log \pi(\mathbf{u}_t^1|\mathbf{x}_t^1) & 1 \\ \vdots & \vdots \\ \sum_{t=0}^{N-1} \nabla_\theta \log \pi(\mathbf{u}_t^s|\mathbf{x}_t^s) & 1 \end{bmatrix}$$

2 $$\mathbf{Y} = \begin{bmatrix} \sum_{t=0}^{N-1} r_t^1 & \cdots & \sum_{t=0}^{N-1} r_t^s \end{bmatrix}^T$$

Compute gradient and expected return

3 $$\begin{bmatrix} \delta\theta \\ J \end{bmatrix} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{Y}.$$

output: natural gradient estimate $\mathbf{g} = \delta\theta$

D. Improved Policy Updates using Rprop

The basic principle of Rprop is to eliminate the potentially harmful influence of the magnitude of the partial derivative on the parameter update [10]. As a consequence, only the sign of the derivative is considered to indicate the direction of the parameter update. The size of the parameter change is exclusively determined by a so-called 'update-value' $\Delta_i^{(t)}$, which is an individual, time-varying value for each parameter i.

$$\theta_i^{(t+1)} = \begin{cases} \theta_i^{(t)} + \Delta_i^{(t)} & , \text{ if } (\nabla_\theta J(\theta))_i^{(t)} > 0 \\ \theta_i^{(t)} - \Delta_i^{(t)} & , \text{ if } (\nabla_\theta J(\theta))_i^{(t)} < 0 \\ 0 & , \text{ else} \end{cases}$$

where $(\nabla_\theta J(\theta))_i^{(t)}$ denotes the $i-th$ component of the gradient $\nabla_\theta J(\theta)$ at time step t.

The second step of Rprop learning is to determine the new update-values $\Delta_i(t)$. This is based on a sign-dependent adaptation process.

$$\Delta_i^{(t)} = \begin{cases} \eta^+ * \Delta_i^{(t-1)} & , \text{ if } (\nabla_\theta J(\theta))_i^{(t-1)} * (\nabla_\theta J(\theta))_i^{(t)} > 0 \\ \eta^- * \Delta_i^{(t-1)} & , \text{ if } (\nabla_\theta J(\theta))_i^{(t-1)} * (\nabla_\theta J(\theta))_i^{(t)} < 0 \\ \Delta_i^{(t-1)} & , \text{ else} \end{cases} \qquad (9)$$

$$\text{where } 0 < \eta^- < 1 < \eta^+$$

In other words, the adaptation-rule works as follows: Every time the partial derivative of the corresponding parameter changes its sign, which indicates that the last update was too big and the algorithm has jumped over a local minimum, the update-value Δ_i is decreased by the factor η^-. If the derivative

retains its sign, the update-value is slightly increased in order to accelerate convergence in shallow regions.

In order to reduce the number of freely adjustable parameters, often leading to a tedious search in parameter space, the increase and decrease factor are set to fixed values, i.e. $\eta^- = 0.5$ and $\eta^+ = 1.2$.

At the beginning of learning, the update values Δ_i are set to an initial value, Δ_0. Typically, $\Delta_0 = 0.1$, is used as a default value. To assure that the update values stay in a reasonable range, the update values are restricted to fulfill $\Delta_{min} \leq \Delta_i \leq \Delta_{max}$. For the following experiments, the standard choice was $\Delta_{min} = 0.01$ and $\Delta_{max} = 5.0$.

III. Setup

In this section, we outline the main parts of the setup, i.e., (i) the plant, rewards and policy as well as (ii) the learning setup.

A. Plant, Rewards & Policy:

We focussed on the cart-pole system as a well-understood plant using the standard dynamics and standard parameters (see Appendix VI). As the control problem definition, we used a regulator task, that is, we do not only require the controller to avoid failure of the pole, but we want the controller to move the cart to the target position in the middle of the track with the pole standing upright.

The reward signal basically follows the definition given in [12]: the target set for the state is reached, when the angle of the pole is within $[-0.05\text{rad}, +0.05\text{rad}]$ and the position of the cart is within $[-0.05m, +0.05m]$. The system failed, if the absolute pole angle is larger than 0.7 or the absolute cart position is larger than $2.4m$. In case of a failure, the episode is stopped. The final reward is computed by $-2*(N-t)$, where N gives the maximum episode length (here: $N = 200$ for training and $N = 500$ for test), and t is the time step, where the failure occured. This means, that a later failure is better than an early failure. In case of the state being within the target region, the reward is 0 and the episode is continued (since the system might leave the target region again). In every other situation, the reward is -1. The above setting expresses the desire for a minimum-time controller, that ends up in the target region, with - in case of an inevitable failure - the preference of a longer balancing time over a short one.

The controller was represented using a linear parameterized policy

$$u = \theta \mathbf{x} + \epsilon, \tag{10}$$

with policy parameters θ where $\epsilon = 0$ for deterministic policies and $\epsilon \sim \mathcal{N}(0, \sigma^2)$ for stochastic policies. Policy gradient methods typically require that the learning process is started with a reasonable policy. Such initial conditions were realized by selecting the parameters randomly while ensuring that these lead to an overall stable system. Additionally, in Section IV-E, we tested the methods when started with arbitrary control policies. The VPG and the NG method require a stochastic policy which explores; this was realized by adding a zero mean gaussian noise with variance 1 to the control signal. The FD method relies on parameter perturbations which are selected using uniformly distributed random number in the range $[-\delta\theta_{\max}, +\delta\theta_{\max}]$ with $\delta\theta_{\max}$ set to 2.0 (unless stated differently).

B. Learning Setup

We evaluated three different methods for gradient estimation, i.e., Finite-Differences (FD), Vanilla Policy Gradient (VPG) and Natural Gradient (NG), as described in Section II. Each of these methods was combined with both simple gradient descent and the Rprop method for updating the controller parameters. The parameters for the learning rates for gradient descent where adapted manually for each task. Rprop uses the standard values of $\Delta_{max} = 5$, $\Delta_{min} = 0.01$ and $\Delta_0 = 0.1$ for all tasks.

In all experiments, the policy was updated after every 10 episodes. Performance is measured after every 50 episodes by doing 50 test runs starting from random initial states where the pole angle is between -0.2rad and 0.2rad and the cart position is within -0.5m and 0.5m. For testing, the gaussian noise in the policy was set to 0. Every experiment was repeated 20 times, each with randomly determined (but stabilizing) initial controller parameters. The figures report the average cumultated costs over these 20 runs.

To summarize the learning results in a tabular representation, we distinguish between two performance levels of the learned controllers: reaching average cumulated costs of -120 (corresponding to a satisfying controller performance), and reaching average cumulated costs of -80 (corresponding to a good controller performance). The table reports the number of episodes that are needed to reach the respective performance level and the overall best result achived in a maximum of 50,000 episodes. The average cumulated cost value of the initial controllers is -245.7.

C. Illustrative Learning Result

Figure 2 illustrates the results of a typical successful learning trial. The learning controller is started with an initial policy that has an average cumulated cost value of -384. Applying the initial policy to a sample initial state, the position of the cart reaches the target position only very slowly after about 700 cycles (which corresponds to about 14s in realtime). After learning, the controller achieves an average cumulated cost value of -89. Accordingly, control is much faster with the learned controller: after only about 200 cycles (corresponding to 4s), the cart stabily reached the target position with the pole standing upright (the latter is not shown in the figure). In this example case, the controller was trained by VPG-Rprop within 1500 episodes. Also note, that the target position is achieved with a very high precision.

IV. Results

We compare the results of the main gradient estimators described in Section II when applied in the setup described in Section III. Each time, we compare the setup using standard gradient descent with the Rprop rule.

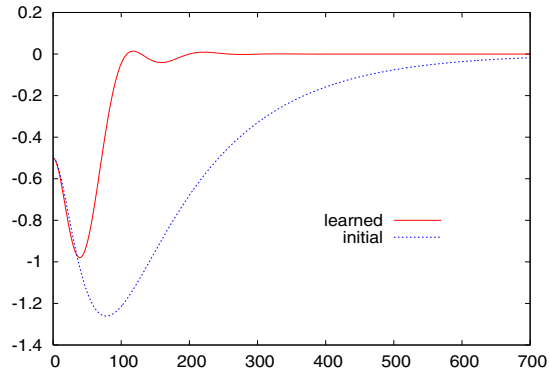

Fig. 2. Comparison of control behavior of initial controller (value -384) and controller learned by vpg Rprop in 1500 episodes (value -89). The figure shows the position of the cart over the number of time steps (0.02s each). Both controllers accurately control the position to the final position. The trained controller achieves the position much faster. In both cases, the pendulum is standing upright at the end of the episodes.

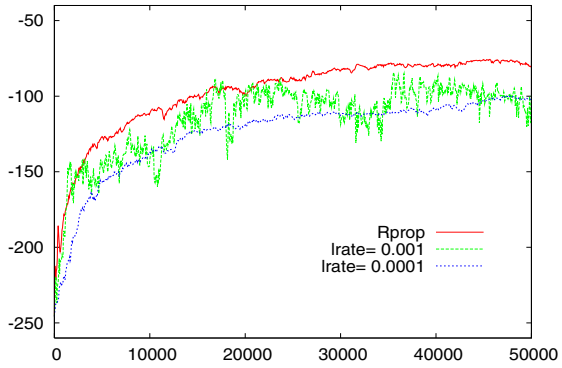

Fig. 3. Finite Difference method combined with both gradient descent and Rprop. The figure shows average cumulated costs over the number of training episodes. Using standard gradient descent, a large learning rate might lead to oscillating learning behaviour, whereas small learning rates might result in slow learning progress. Rprop achieves a fast and smooth learning behaviour with standard parametrisation.

A. Estimating the Gradient using Finite Differences (FD)

1) FD and Standard Gradient Descent (FD-GD): The combination of the FD method to estimate the gradient and the standard gradient descent rule to update the controller parameters delivered controllers with satisfying performance. When the learning rate was carefully tuned ($\epsilon = 0.001$), the performance measure of the controller could be increased from an average cumulated cost of about -250 to -84.3 (row 2 in Table I), which corresponds to a good controller performance. A satisfying controller with a performance value better than -120 was reached for the first time after 12300 episodes.

When taking a closer look at the learning curve for $\epsilon = 0.001$, a highly varying controller performance can be observed. This is a typical indicator, that the learning rate for gradient descent might have be too large. Indeed, when choosing a smaller value $\epsilon = 0.0001$, the learning behaviour was smoother but also significantly slower. This trend is continued, when the learning rate is lowered further (see figure 3). Increasing the learning rate beyond 0.001 resulted in a non-successful learning behaviour (row 1 in table I).

2) FD and Rprop (FD-Rprop): Rprop adapts the size of the update step based on the observed curvature of the cost function. When the gradient information is reliable, this usually leads to a faster and more robust optimization process. Combining Rprop with the FD method actually resulted in a significant improvement compared to standard gradient descent.

TABLE I
RESULTS WITH THE FD METHOD

	> -120	> -80	best
FD-GD, $\epsilon = 0.01$	-	-	-245.7
FD-GD, $\epsilon = 0.001$	12 300	-	-84.3
FD-GD, $\epsilon = 0.0001$	18 100	-	-99.7
FD-GD, $\epsilon = 0.00001$	-	-	-158.4
FD-Rprop	7 450	45 650	-75.6

FD-Rprop on average reaches a satisfying controller (better than -120) within 7450 episodes and a good one (with a cost of less than -80) after 45 650 episodes (see row 4 in Table I). The final value reached was -75.6, which is considerably better than the value reached by the FD-GD approach. Also, learning behaviour was much smoother than with the most successful but aggressive gradient descent parameter (see Figure 3).

An additional benefit of using Rprop rule is that it works well with the standard parametrization given in Section III. Being less sensitive to the choice of its parameters is an highly important feature when it comes to more complex real-world applications.

3) Influence of further FD Parameters: We initially expected the FD method to be more sensitive to noise in the learning process, e.g. induced by selecting random initial states or taking random perturbation steps). However, we found that in our experiments the FD method behaved surprisingly robust in the presence of that kind of noise; a fact that is additionally supported by the results on the application to a noisy plant (see Section IV-D).

FD also turned out to be very robust against the choice of the maximum perturbation step size parameter. Values between 0.1 and 10 lead to very similar results. Only when the value was chosen extremely big (e.g larger than 50), the learning curves started to become less smooth.

B. Estimating the Gradient by Vanilla Policy Gradient (VPG)

1) VPG and Standard Gradient Descent (VPG-GD): Combining vanilla policy gradient and standard gradient descent for updating the parameters, a value of 0.2 for the learning rate gave the best results (Row 2 in Table II). The results improved over the FD results in the previous section. Not only did it learn much faster (only 1200 episodes to learn a satisfying controller with performance better than -120), but also achieved a much better controller performance (of roughly -75.8).

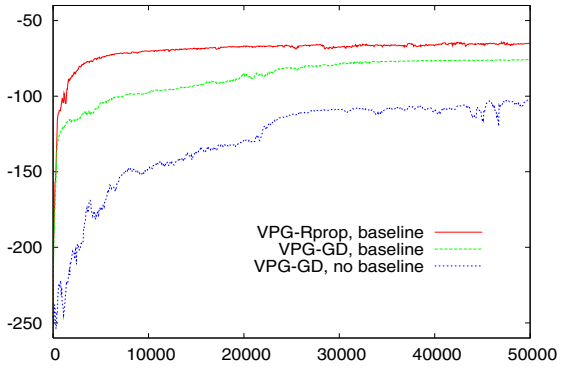

Fig. 4. Vanilla Policy Gradient (VPG) method combined with both gradient descent and Rprop. Figure shows average cumulated costs over the number of training episodes. Blue line shows gradient descent behaviour, when no baselines are used. Using optimal baselines, gradient descent performs much better (green line). Rprop and baselines results in a very fast and effective learning behaviour.

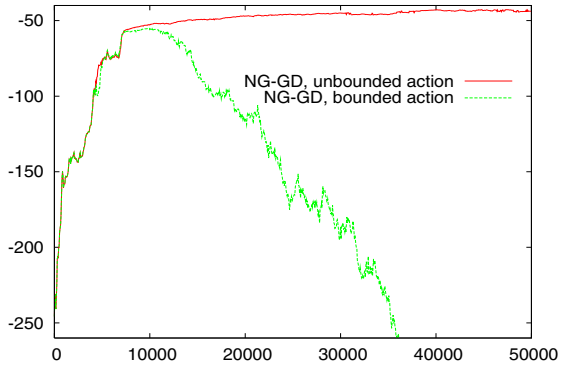

Fig. 5. NG and standard gradient descent, prior control. The performance finally decreases again. Only when the control signal is not constrained, a continuously improving performance was achieved.

Varying the learning rate by a factor of 10 has a considerable effect on the results of learning with respect to speed and final performance (see Rows 1 and 3 in Table II). With a too large learning rate, we observed that learning got trapped very early on a low but stable level. In this case, it seems that the variance of the stochastic policy has decreased too fast which results in bad exploration behaviour.

2) VPG and Rprop (VPG-Rprop): Vanilla policy gradient and Rprop were found to be a very successful combination (see row 4 in Table II). Not only good controllers were found quickly (a performance better than -120 in 450 episodes and better than -80 in 3000 episodes), but also a final performance achieved -64.3 was very good. This speed is on average more than 10 time steps faster than the best controller found by VPG-GD.

Again, for Rprop the standard parameters given in Section III were used.

3) The Effect of Baselines on VPGs: To examine the influence of the baseline (see Section II-B), we conducted also experiments with a baseline of $b_t^i = 0$, i.e., no baseline. Using no baseline at all has a dramatic effect on VPG. In combination with gradient descent learning, it became much slower and the resulting controllers performed significantly worse (see rows 5-7 in Table II and Figure 4). When no baselines are used, the Rprop update did not work at all.

TABLE II

RESULTS WITH THE VPG METHOD

	> -120	> -80	best
VPG-GD, baseline, $\epsilon = 0.02$	10 850	-	-102.5
VPG-GD, baseline, $\epsilon = 0.2$	1 200	26 450	-75.8
VPG-GD, baseline, $\epsilon = 2.0$	-	-	-154.2
VPG-Rprop, baseline	450	3 000	-64.3
VPG-GD, no baseline, $\epsilon = 0.003$	-	-	-123.8
VPG-GD, no baseline, $\epsilon = 0.03$	22 200	-	-102.4
VPG-GD, no baseline, $\epsilon = 0.3$	-	-	-217.7
VPG-Rprop, no baseline	-	-	-245.7

We assume, that this is due to the increased variance of the gradient estimates without baselines: in this case, the gradient information is too noisy and the Rprop step-size adaptation can not work effectively. Only when we increased the number of samples per update (which also decreases the variance in the gradient information), we could make Rprop to work without baselines. But of course, since more episodes are needed to do a parameter update, this method results in a slower learning process.

C. Estimating the Gradient by the Natural Gradient Method (NG)

1) NG and Standard Gradient Descent (NG-GD): The combination of natural gradients and standard gradient descent resulted both in reasonably fast and successful learning (see rows 1-3 in Table III). A good controller (better than -80) could be learned in 5 050 episodes, which is about 5 times faster than the combination of vanilla policy gradient and gradient descent. Also NG-GD was able to produce the best controller obtained so far (average reward of -55).

An interesting effect we observed, is that the performance of NG-GD decreased again after continuing the learning process over a certain limit. Varying the parameters (learning-rate, number of samples per update, variance) did not solve the problem. The cause of this phenomenon lies in the fact, that the control signal actually applied to the plant is cut off at $\pm 10N$: if we allow arbitrary control signals, then the learning curve continuously improves as expected (see Figure 5). This artifact will be further investigated in the future.

2) NG and Rprop (NG-Rprop): Combining NG and Rprop worked but could not compete with standard gradient descent

TABLE III

RESULTS WITH THE NG METHOD

	> -120	> -80	best
NG-GD, $\epsilon = 0.001$	14 400	37 350	-62.1
NG-GD, $\epsilon = 0.01$	3750	5 050	-55.0
NG-GD, $\epsilon = 0.1$	-	-	-245.7
NG-Rprop, $\epsilon = 0.01$	-	-	-129.5

when used with good learning rates (line 4 in Table III). NG-Rprop was able to produce controllers with an average performance of -129.5, which is an improvement of the initial performance, but far from the performance achieved with other methods. Looking closer at the results, we saw, that for some runs, learning was fast and effective, while other runs failed completely. On average, this resulted into bad performance. Future work might explore, whether a refined approach will result in better and more robust learning behaviour.

D. Application to a Noisy Plant

To check the behaviour of the learning methods in a noisy environment, gaussian noise was added to the plant state variables in every time step (see appendix). The noise-level parameter was set to 0.05. The average performance value of the intial controllers was -809.7

The initial conjecture, that the FD method is particularly sensitive to noise and probably would not work at all, was not confirmed. Both the gradient descent and the Rprop variant achieved to produce satisfying controllers, with the Rprop variant learning faster and producing better results (row 1 and 2 in table IV).

VPG reached an overall better performance level than FD, VPG-Rprop being again faster than gradient descent (row 3 and 4 in table IV). The Rprop-Version of VPG reached a very high performance level very quickly. NG-GD reached the best overall performance, whereas NG-Rprop managed to improve the initial controller performance, but did not deliver satisfying results (row 5 and 6 in table IV).

E. Learning in the Absence of Prior Knowledge

Policy gradient methods in general make the assumption that a fair amount of prior knowledge about the policy is available at the beginning of learning and given in form of the (initial) parameterized policy. In the cart-pole example, both the general form of the policy law (a linear combination of state variables) and on the initial policy parameters was used so far.

This experiment tests how the methods behave in the absence of knowledge on reasonable initial controller parameters.

	> -700	> -650	> -600	best
FD-GD	35 050	-	-	679.9
FD-Rprop	10 050	25 350	-	-633.57
VPG-GD	21 750	-	-	-678.0
VPG-Rprop	350	1 650	10 650	-577.87
NG-GD	2 100	3 950	9 450	-528.2
NG-Rprop	-	-	-	-758.0

TABLE IV

APPLICATION OF THE VARIOUS METHODS TO A NOISY PLANT

TABLE V

LEARNING, WHEN CONTROLLER PARAMETERS ARE INITIALIZED WITH ZERO

	> -120	> -80	best
FD-Rprop	22 900	46 950	-78.8
all other	-	-	-1000

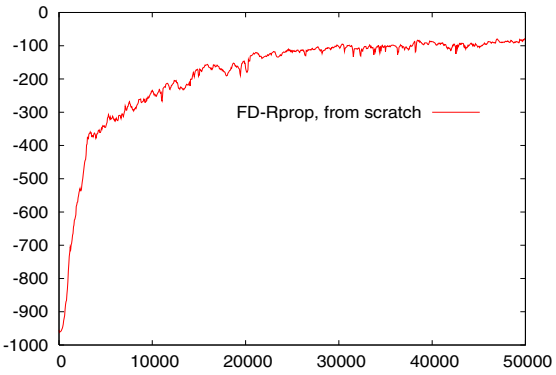

Fig. 6. Finite Differences in combination with Rprop were the only method that found a controller when starting with no initial knowledge on the controller parameters (ie. $R = (0, 0, 0, 0)^T$).

To test this, controller parameters were initialized with 0 each. Clearly, this controller initially is not able to fulfill the control task.

The only method that shows improvement at all was the finite difference approach combined with Rprop (Figure 6). However, in order to get a fast learning and good performance, we had to choose quite aggressive learning meta-parameters. The maximum perturbation step was chosen to be 50.0 and the minimum parameter update step was $\Delta_{min} = 0.1$. Using the original parameters showed improvement, but resulted in a very slow learning process.

In contrast, both VPG and NG methods did not show any improvement at all over a wide range of tested parameters, both with gradient descent and Rprop.

In a second, more difficult setting, controller parameters were randomly initialized to each lie within the interval $[-50, 50]$. This is a particularly hard test, since controller parameters might be far from reasonable or even good values. Again, FD + Rprop was the only method that was able to produce reasonable controllers even out of a very bad initial situation.

F. Summary

A central focus of this study was dedicated to the situation, when a reasonable parametrisation of initial controllers is available. This is the typical assumption when policy gradient methods are to be applied. For the cart-pole benchmark considered here, all of the gradient computation methods (FD, VPG, NG) where able to successfully improve initial controller performance when combined with gradient descent parameter update. However, in order to be successful, a careful tuning of the learning rate is necessary. A good value for the learning rate differs from application to application, potentially by several magnitudes. If the learning rate is too large, no learning occurs, and even a decrease of the controller performance was observed. It therefore seems reasonable, to start with a rather small value first and if learning is successful, increase it, until a good value is found. For FD and VPG, the Rprop method to update the parameter values resulted in a siginificant

improvement of the results, both with respect to learning speed and performance of the resulting controllers. Also, standard parameters can be used, which might be an important fact when it comes to practical real world applications, where tedious parameter tuning might be costly.

NG and VPG turned out to be considerably faster and to result in better controllers than FD. FD on the contrary has the advantage, that it is the simplest algorithm of all, and that it might also be applied in more general situations (e.g. it does not rely on a stochastic policy). FD-Rprop also was the only algorithm, that was able to learn a controller in the absence of a good guess for the controller parameters. NG was superior to VPG with respect to the final controller performance learned. When VPG was combined with Rprop, it was faster than NG with gradient descent, but could not reach the very good performance that NG-GD achieved. The baseline method turned out to be very efficient for the VPG method both with respect to learning time and to the controller performance that could be achieved.

V. Conclusion

The paper discusses three main algorithms of policy gradient methods on the cart-pole regulator benchmark. Several variants of the original methods are discussed. The results obtained in the empirical section may serve as a baseline for the further investigation of modifications and improvements of the algorithms. To make comparison easier, complete source code of plant and controllers is released, available under www.ni.uos.de/pgmethods. Since only very general interfaces are used, it should be easy to integrate the code into other RL software packages, like e.g. RL-Glue developped at University of Alberta.

References

[1] S. Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10, 1998.

[2] G. Endo, J. Morimoto, T. Matsubara, J. Nakanishi, and G. Cheng. Learning cpg sensory feedback with policy gradient for biped locomotion for a full-body humanoid. In *AAAI 2005*, 2005.

[3] M. C. Fu. Feature article: Optimization for simulation: Theory vs. practice. *INFORMS Journal on Computing*, 14(3):192–215, 2002.

[4] V. Gullapalli, J. Franklin, and H. Benbrahim. Aquiring robot skills via reinforcement learning. *IEEE Control Systems*, -(39), 1994.

[5] S. A. Kakade. Natural policy gradient. *Advances in Neural Information Processing Systems 14*, 2002.

[6] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, New Orleans, LA, May 2004.

[7] A. Y. Ng and M. Jordan. Pegasus: A policy search method for large mdps and pomdps. In *Uncertainty in Artificial Intelligence, Proceedings of the Sixteenth Conference*, 2000.

[8] J. Peters and S. Schaal. Policy gradient methods for robotics. In *Proceedings of the ieee international conference on intelligent robotics systems (iros 2006)*, 2006.

[9] J. Peters, S. Vijayakumar, and S. Schaal. Natural actor-critic. In *Proceedings of the 16th european conference on machine learning (ecml 2005)*, pages 280–291. springer, 2005.

[10] M. Riedmiller. Advanced supervised learning in multi-layer perceptrons - from backpropagation to adaptive learning algorithms. *Int. Journal of Computer Standards and Interfaces*, 16:265–278, 1994. Special Issue on Neural Networks.

[11] M. Riedmiller, R. Hafner, S. Lange, and S. Timmer. Clsquare - a closed loop simulation system.

[12] M. Riedmiller, M. L. Littman, M. G. Lagoudakis, N. Vlassis, S. Whiteson, and A. White. Reinforcement learning benchmarks and bake-offs i, ii. Workshop at the 2005 Neural Information Processing Systems (NIPS) Conference, Decembre 2005.

[13] R. Tedrake, T. W. Zhang, and H. S. Seung. Learning to walk in 20 minutes. In *Proceedings of the Fourteenth Yale Workshop on Adaptive and Learning Systems*, Russ Tedrake, Teresa Weirui Zhang, and H. Sebastian Seung. (2005) Learning to Walk in 20 Minutes. In Proceedings of the Fourteenth Yale Workshop on Adaptive and Learning Systems, Yale University, New Haven, CT, 2005, 2005.

VI. Appendix

The dynamics of the cart-pole system are given by the following equations:

$$\ddot{x} = \frac{F - m_p\, l\, (\ddot{\theta}\, \cos\theta - \dot{\theta}^2\, \sin\theta)}{m_c + m_p}$$

$$\ddot{\theta} = \frac{g\, \sin\theta\, (m_c + m_p) - (F + m_s\, l\, \dot{\theta}^2\, \sin\theta) \cos\theta}{\frac{4}{3} l\, (m_c + m_p) - m_p\, l\, \cos^2\theta}$$

where $l = 0.5$ (half length of pole), $m_c = 1.0$ (mass of cart), $m_c = 0.1$ (mass of pole), $g = 9.81 m/s^2$ (gravity), and $-10N \leq F \leq +10N$ (force applied to the cart). The control interval was 0.02s. The dynamics were simulated by a fourth order Runge-Kutta method.

For the noisy plant, a noise vector ξ was added to the state vector $\mathbf{x} = (\theta, \dot{\theta}, s, \dot{s})^T$ in every time-step, i.e. $\mathbf{x}(\mathbf{t}+\mathbf{1}) := f(\mathbf{x}(\mathbf{t}), u(t)) + \xi$. ξ is a four dimensional, zero-mean, gaussian distributed noise vector, i.e. $\xi \sim N(\mathbf{0}, \Sigma)$, with $\Sigma = noise\_level * diag(0.5, 2., 2., 2.)$.