Robot Learning with State-Dependent Exploration

Thomas Rückstieß, Martin Felder, Frank Sehnke, Jürgen Schmidhuber

Abstract— **Policy gradient algorithms are among the few learning methods successfully applied to demanding real-world problems including those found in the field of robotics. While Likelihood Ratio (LR) methods are typically used to estimate the gradient, they suffer from high variance due to random exploration at each timestep during the rollout. We therefore evaluate several policy gradient methods with state-dependent exploration (SDE), a recently introduced alternative to random exploration, which deterministically returns the same action for a given state during one episode. We apply SDE to a simulated robotics task with realistically modelled physics, and compare it to random exploration within several different learning schemes. Our experiments show that SDE outperforms traditional random exploration in almost every case.**

I. Introduction

Reinforcement Learning (RL) is a powerful concept for dealing with semi-supervised control tasks, since they don't require a teacher to tell the agent the correct action for a given situation. While exploring the space of possible actions, the reinforcement signal can be used to adapt the parameters governing the agent's behavior. Classical RL algorithms [1], [2] are designed for problems with a limited, discrete number of states. For these scenarios, sophisticated exploration strategies can be found in the literature [3], [4].

In contrast, Policy Gradient (PG) methods as pioneered by Williams [5] can deal with continuous states and actions, as they appear in many real-life settings. They can handle function approximation, avoid sudden discontinuities in the action policy during learning, and were shown to converge at least locally [6]. Successful applications are found e.g. in robotics [7], [8], [9], financial data prediction [10] or network routing [11].

However, a major problem in RL remains that feedback is rarely available at every time step. Imagine a robot trying to exit a labyrinth within a set time, with a default policy of driving straight. Feedback is given at the end of an *episode*, based on whether it was successful or not. PG methods most commonly use a random exploration strategy [5], [7], where the deterministic action ("if wall ahead, go straight") at each time step is perturbed by Gaussian noise. This way, the robot may wiggle free from time to time, but it is very hard to improve the policy based on this success, due to the high variance in the gradient estimation. Obviously, a lot of research has gone into devising smarter, more robust ways of estimating the gradient, as detailed in the excellent survey by Peters [7].

Our novel approach, introduced in [12], is much simpler and targets the exploration strategy instead: In the example, the robot would use a deterministic function providing an exploration offset consistent throughout the episode, but still depending on the state. This might easily change the policy into something like "if wall ahead, veer a little left", which is much more likely to lead out of the labyrinth, and thus can be identified easily as a policy improvement. Hence, our method, which we call *state-dependent exploration* (SDE), causes considerable variance reduction and therefore faster convergence. Because it only affects exploration and does not depend on a particular gradient estimation technique, SDE can be enhanced with any episodic likelihood ratio (LR) method, like REINFORCE [5], GPOMDP [13], or ENAC [14], to reduce the variance even further.

Our exploration strategy is in a sense related to Finite Difference (FD) methods like SPSA [15] or the recently developed PGPE [16], as both create policy deltas (or strategy variations) rather than perturbing single actions. However, direct parameter perturbation has to be handled with care, since small changes in the policy can easily lead to unforseen and unstable behavior and a fair amount of system knowledge is therefore necessary. Furthermore, FD are very sensitive to noise and hence not suited for many real-world tasks. SDE does not suffer from these drawbacks—it embeds the power of FD exploration into the stable LR framework.

The remainder of this paper is structured as follows: Section II briefly introduces the policy gradient framework. Our novel exploration strategy SDE will be explained in detail in section III. Experiments and their results are described in section IV. The paper concludes with a short discussion in section V.

II. Policy Gradient Framework

An advantage of policy gradient methods is that they don't require the environment to be Markovian, i.e. each controller action may depend on the whole history encountered. So we will introduce our policy gradient framework for general non-Markovian environments but later assume a Markov Decission Process (MDP) for ease of argument.

A. General Assumptions

A policy $\pi(u|h,\theta)$ is the probability of taking action u when encountering history h under the policy parameters θ. Since we use parameterized policies throughout this paper, we usually ommit θ and just write $\pi(u|h)$. We will use h^π for the history of all the observations x, actions u, and rewards r encountered when following policy π. The history at time $t = 0$ is defined as the sequence $h_0^\pi = \{x_0\}$, consisting only of the start state x_0. The history at time t consists of all the observations, actions and rewards encountered so far and is defined as $h_t^\pi = \{x_0, u_0, r_0, x_1, \ldots, u_{t-1}, r_{t-1}, x_t\}$.

The return for the controller whose interaction with the environment produces history h^π is written as $R(h^\pi)$, which is defined as $R(h^\pi) = a_\Sigma \sum_{t=0}^{T} a_D\, r_t$ with $a_\Sigma = (1-\gamma)$, $a_D = \gamma^t$ for discounted (possibly continuous) tasks and $a_\Sigma = 1/T$, $a_D = 1$ for undiscounted (and thus necessarily episodic) tasks. In this paper, we deal with episodic learning and therefore will use the latter definition. The expectation operator is written as $E\{\cdot\}$.

The overall performance measure of policy π, independent from any history h, is denoted $J(\pi)$. It is defined as $J(\pi) = E\{R(h^\pi)\} = \int p(h^\pi)R(h^\pi)\, dh^\pi$. Instead of $J(\pi)$ for policy π parameterized with θ, we will also write $J(\theta)$.

To optimize policy π, we want to move the parameters θ along the gradient of J to an optimum with a certain learning rate α:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi). \tag{1}$$

The gradient $\nabla_\theta J(\pi)$ is

$$\nabla_\theta J(\pi) \;=\; \int_{h^\pi} \nabla_\theta p(h^\pi) R(h^\pi)\, dh^\pi. \tag{2}$$

B. Likelihood Ratio Methods

Rather than perturbing the policy directly, as it is the case with FD methods [15], [7], LR methods [5] perturb the resulting action instead, leading to a stochastic policy (which we assume to be differentiable with respect to its parameters θ), such as

$$u = f(h, \theta) + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{3}$$

where f is the controller and ϵ the exploration noise. Unlike FD methods, the new policy that leads to this behavior is not known and consequently the difference quotient

$$\frac{\partial J(\theta)}{\partial \theta_i} \;\approx\; \frac{J(\theta + \delta\theta) - J(\theta)}{\delta\theta_i} \tag{4}$$

can not be calculated. Thus, LR methods use a different approach in estimating $\nabla_\theta J(\theta)$, the most basic one being Williams' REINFORCE gradient estimation [5], which makes use of *Monte-Carlo sampling*:

$$\nabla_\theta J(\pi) \;\approx\; \frac{1}{N} \sum_{h^\pi} \sum_{t=0}^{T-1} \nabla_\theta \log \pi(u_t | h_t^\pi) R(h^\pi). \tag{5}$$

A more detailed derivation of the general idea of likelihood ratio policy gradients was presented in [12].

Several approaches to improve gradient estimates are available, as mentioned in the introduction. Neither these nor ideas like baselines [5], the PEGASUS trick [17] or other variance reduction techniques [18] are treated here. They are complementary to our approach, and their combination with SDE will be covered by a future paper.

C. Application to Function Approximation

Here we describe how the results above, in particular (5), can be applied to general parametric function approximation. Because we are dealing with multi-dimensional states $\boldsymbol{x}$ and multi-dimensional actions $\boldsymbol{u}$, we will now use bold font for (column) vectors in our notation for clarification.

To avoid the issue of a growing history length and to simplify the equations, we will assume the world to be Markovian for the remainder of the paper, i.e. the current action only depends on the last state encountered, so that $\pi(u_t|h_t^\pi) = \pi(u_t|x_t)$. But due to its general derivation, the idea of SDE is still applicable to non-Markovian environments.

The most general case would include a multi-variate normal distribution function with a covariance matrix $\boldsymbol{\Sigma}$, but this would square the number of parameters and required samples. Also, differentiating this distribution requires calculation of $\boldsymbol{\Sigma}^{-1}$, which is time-consuming. We will instead use a simplification here and add independent uni-variate normal noise to each element of the output vector seperately. This corresponds to a covariance matrix $\boldsymbol{\Sigma} = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$.[1] The action $\boldsymbol{u}$ can thus be computed as

$$\boldsymbol{u} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta}) + \boldsymbol{e} = \begin{bmatrix} f_1(\boldsymbol{x}, \boldsymbol{\theta}) \\ \vdots \\ f_n(\boldsymbol{x}, \boldsymbol{\theta}) \end{bmatrix} + \begin{bmatrix} e_1 \\ \vdots \\ e_n \end{bmatrix} \tag{6}$$

with $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots]$ being the parameter vector and f_j the jth controller output element. The exploration values e_j are each drawn from a normal distribution $e_j \sim \mathcal{N}(0, \sigma_j^2)$. The policy $\boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x})$ is the probability of executing action $\boldsymbol{u}$ when in state $\boldsymbol{x}$. Because of the independence of the elements, it can be decomposed into $\boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x}) = \prod_{k \in \mathbb{O}} \pi_k(u_k|\boldsymbol{x})$ with $\mathbb{O}$ as the set of indices over all outputs, and therefore $\log \boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x}) = \sum_{k \in \mathbb{O}} \log \pi_k(u_k|\boldsymbol{x})$. The element-wise policy $\pi_k(u_k|\boldsymbol{x})$ is the probability of receiving value u_k as kth element of action vector $\boldsymbol{u}$ when encountering state $\boldsymbol{x}$ and is given by

$$\pi_k(u_k|\boldsymbol{x}) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{(u_k - \mu_k)^2}{2\sigma_k^2}\right), \tag{7}$$

where we substituted $\mu_k := f_k(\boldsymbol{x}, \boldsymbol{\theta})$. We differentiate with respect to the parameters θ_j and σ_j:

$$\frac{\partial \log \boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x})}{\partial \theta_j} \;=\; \sum_{k \in \mathbb{O}} \frac{(u_k - \mu_k)}{\sigma_k^2} \frac{\partial \mu_k}{\partial \theta_j} \tag{8}$$

$$\frac{\partial \log \boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x})}{\partial \sigma_j} \;=\; \frac{(u_j - \mu_j)^2 - \sigma_j^2}{\sigma_j^3} \tag{9}$$

For the linear case, where $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta}) = \boldsymbol{\Theta}\boldsymbol{x}$ with the parameter matrix $\boldsymbol{\Theta} = [\theta_{ji}]$ mapping states to actions, (8) becomes

$$\frac{\partial \log \boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x})}{\partial \theta_{ji}} = \frac{(u_j - \sum_i \theta_{ji} x_i)}{\sigma_j^2} x_i. \tag{10}$$

An issue with nonlinear function approximation (NLFA) is a parameter dimensionality typically much higher than their output dimensionality, constituting a huge search space for FD methods. However, in combination with LR methods, they are interesting because LR methods only perturb the

[1] A further simplification would use $\boldsymbol{\Sigma} = \sigma \boldsymbol{I}$ with $\boldsymbol{I}$ being the unity matrix. This is advisable if the optimal solution for all parameters is expected to lay in similar value ranges.

resulting outputs and not the parameters directly. Assuming the NLFA is differentiable with respect to its parameters, one can easily calculate the log likelihood values for each single parameter.

The factor $\frac{\partial \mu_k}{\partial \theta_j}$ in (8) describes the differentiation through the function approximator. It is convenient to use existing implementations, where instead of an error, the log likelihood derivative with respect to the mean, i.e. the first factor of the sum in (8), can be injected. The usual backward pass through the NLFA—known from supervised learning settings—then results in the log likelihood derivatives for each parameter [5].

III. State-Dependent Exploration

As indicated in the introduction, adding noise to the action u of a stochastic policy (3) at each step enables random exploration, but also aggravates the credit assignment problem: The overall reward for an episode (also called *return*) cannot be properly assigned to individual actions because information about which actions (if any) had a positive effect on the return value is not accessible.[2]

Our alternative approach adds a *state-dependent offset* to the action at each timestep, which can still carry the necessary exploratory randomness through variation between episodes, but will always return the same value in the same state within an episode. We define a function $\hat{\boldsymbol{\epsilon}}(\boldsymbol{x}; \hat{\boldsymbol{\theta}})$ on the states, which will act as a pseudo-random function that takes the state $\boldsymbol{x}$ as input. Randomness originates from parameters $\hat{\boldsymbol{\theta}}$ being drawn from a normal distribution $\hat{\theta}_j \sim \mathcal{N}(0, \hat{\sigma}_j^2)$. As discussed in section II-C, simplifications to reduce the number of variance parameters can be applied. The action is then calculated by

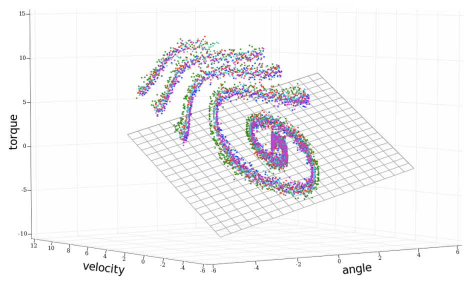
$$\boldsymbol{u} = \boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta}) + \hat{\boldsymbol{\epsilon}}(\boldsymbol{x}; \hat{\boldsymbol{\theta}}), \quad \hat{\theta}_j \sim \mathcal{N}(0, \hat{\sigma}_j^2). \tag{11}$$

If the parameters $\hat{\boldsymbol{\theta}}$ are drawn at each timestep, we have an LR algorithm as in (3) and (6), although with a different exploration variance. However, if we keep $\hat{\boldsymbol{\theta}}$ constant for a full episode, then our action will have the same exploration added whenever we encounter the same state (Figure 1). Depending on the choice of $\hat{\boldsymbol{\epsilon}}(\boldsymbol{x})$, the randomness can further be "continuous", resulting in similar offsets for similar states. Effectively, by drawing $\hat{\boldsymbol{\theta}}$, we actually create a *policy delta*, similar to FD methods. In fact, if both $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\Theta})$ with $\boldsymbol{\Theta} = [\theta_{ji}]$ and $\hat{\boldsymbol{\epsilon}}(\boldsymbol{x}, \hat{\boldsymbol{\Theta}})$ with $\hat{\boldsymbol{\Theta}} = [\hat{\theta}_{ji}]$ are linear functions, we see that

$$\begin{aligned} \boldsymbol{u} &= \boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\Theta}) + \hat{\boldsymbol{\epsilon}}(\boldsymbol{x}; \hat{\boldsymbol{\Theta}}) \\ &= \boldsymbol{\Theta}\boldsymbol{x} + \hat{\boldsymbol{\Theta}}\boldsymbol{x} \\ &= (\boldsymbol{\Theta} + \hat{\boldsymbol{\Theta}})\boldsymbol{x}, \end{aligned} \tag{12}$$

which shows that direct parameter perturbation methods (cf. (4)) are a special case of SDE and can be expressed in this more general reinforcement framework.

[2]GPOMDP [13], also known as the Policy Gradient Theorem [6], does consider single step rewards. However, it still introduces a significant amount of variance to a rollout with traditional random exploration.



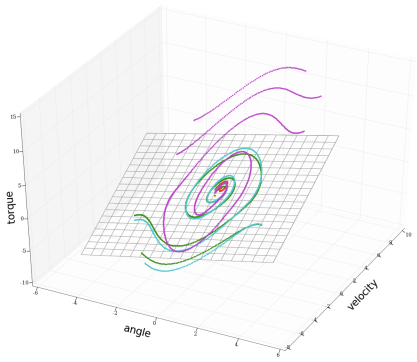


Fig. 1. Illustration of the main difference between random (top) and state-dependent (bottom) exploration. Several rollouts in state-action space of a task with state $\boldsymbol{x} \in \mathbb{R}^2$ (x- and y-axis) and action $u \in \mathbb{R}$ (z-axis) are plotted. While random exploration follows the same trajectory over and over again (with added noise), SDE instead tries different *strategies* and can quickly find solutions that would take a long time to discover with random exploration.

A. Updates of Exploration Variances

For a linear exploration function $\hat{\boldsymbol{\epsilon}}(\boldsymbol{x}; \hat{\boldsymbol{\Theta}}) = \hat{\boldsymbol{\Theta}}\boldsymbol{x}$ it is possible to calculate the derivative of the log likelihood with respect to the variance. We will derive the adaptation for general $\hat{\sigma}_{ji}$, any parameter reduction techniques from II-C can be applied accordingly.

The distribution of the action vector elements is $u_j = f_j(\boldsymbol{x}, \boldsymbol{\Theta}) + \hat{\boldsymbol{\Theta}}_j \boldsymbol{x} = f_j(\boldsymbol{x}, \boldsymbol{\Theta}) + \sum_i \hat{\theta}_{ji} x_i$, with $f_j(\boldsymbol{x}, \boldsymbol{\Theta})$ as the jth element of the return vector of the deterministic controller f and $\hat{\theta}_{ji} \sim \mathcal{N}(0, \hat{\sigma}_{ji}^2)$. Applying the standard properties of normal distributions, we see that the action element u_j is distributed as

$$u_j \sim \mathcal{N}(f_j(\boldsymbol{x}, \boldsymbol{\Theta}), \sum_i (x_i \hat{\sigma}_{ji})^2), \tag{13}$$

where we will substitute $\mu_j := f_j(\boldsymbol{x}, \boldsymbol{\Theta})$ and $\sigma_j^2 := \sum_i (x_i \hat{\sigma}_{ji})^2$ to obtain expression (7) for the policy components again. Differentiation of the policy with respect to the free parameters $\hat{\sigma}_{ji}$ yields:

$$\frac{\partial \log \boldsymbol{\pi}(\boldsymbol{u}|\boldsymbol{x})}{\partial \hat{\sigma}_{ji}} = \frac{(u_j - \mu_j)^2 - \sigma_j^2}{\sigma_j^4} x_i^2 \hat{\sigma}_{ji} \tag{14}$$

For more complex exploration functions, calculating the exact derivative for the sigma adaptation might not be possible and heuristic or manual adaptation (e.g. with slowly decreasing $\hat{\sigma}$) is required.

B. Stochastic Policies

The original policy gradient setup as presented in e.g. [5] conveniently unifies the two stochastic features of the algorithm: the stochastic exploration and the stochasticity of the policy itself. Both were represented by the Gaussian noise added on top of the controller. While elegant on the one hand, it also conceals the fact that there are two different stochastic processes. With SDE, randomness has been taken out of the controller completely and is represented by the seperate exploration function. So if learning is switched off, the controller only returns deterministic actions. But in many scenarios the best policy is necessarily of stochastic nature.

It is possible and straight-forward to implement SDE with stochastic policies, by combining both random and state-dependent exploration in one controller, as in

$$\boldsymbol{u} = f(\boldsymbol{x}; \boldsymbol{\theta}) + \boldsymbol{\epsilon} + \hat{\boldsymbol{\epsilon}}(\boldsymbol{x}; \hat{\boldsymbol{\theta}}), \tag{15}$$

where $\epsilon_j \sim N(0, \sigma_j)$ and $\hat{\theta}_j \sim N(0, \hat{\sigma}_j)$. Since the respective noises are simply added together, none of them affects the derivative of the log-likelihood of the other and σ and $\hat{\sigma}$ can be updated independently. In this case, the trajectories through state-action space would look like a noisy version of Figure 1, bottom.

IV. Experiments

We tested our algorithm on a series of experiments based on a simulated robot hand with realistically modelled physics. We chose this experiment to show the predominance of SDE over random exploration, especially in a realistic robot task. We used the Open Dynamics Engine[3] to model the hand, arm, body, and object. The arm has 3 degrees of freedom: shoulder, elbow, and wrist, where each joint is assumed to be a 1D hinge joint, which limits the arm movements to forward-backward and up-down. The hand itself consists of 4 fingers with 2 joints each, but for simplicity we only use a single actor to move all finger joints together, which gives the system the possibility to open and close the hand, but it cannot control individual fingers. These limitations to hand and arm movement reduce the overall complexity of the task while giving the system enough freedom to catch the ball. A 3D visualization of the robot attempting a catch is shown in Fig. 2.

A. Experiment setup

The information given to the system are the three coordinates of the ball position, so the robot "sees" where the ball is. It has four degrees of freedom to act, and in each timestep it can add a positive or negative torque to the joints. The controller therefore has 3 inputs and 4 outputs. We map inputs directly to outputs, but squash the outgoing signal with a tanh-function to ensure output between -1 and 1.

The reward function is defined as follows: upon release of the ball, in each time step the reward can either be -3 if the ball hits the ground (in which case the episode is considered a

[3]The Open Dynamics Engine (ODE) is an open source physics engine, see `http://www.ode.org/` for more details.

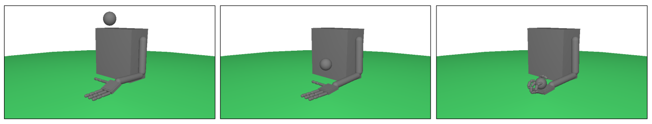


Fig. 2. Visualization of the simulated robot hand while catching a ball. The ball is released 5 units above the palm, where the palm dimensions are 1 x 0.1 x 1 units. When the fingers grasp the ball and do not release it throughout the episode, the best possible return (close to -1.0) is achieved.

failure, because the system cannot recover from it) or else the negative distance between ball center and palm center, which can be any value between -3 (we capped the distance at 3 units) and -0.5 (the closest possible distance considering the palm heights and ball radius). The return for a whole episode is the mean over the episode: $R = \frac{1}{N}\sum_{n=1}^{N} r_t$. In practice, we found an overall episodic return of -1 or better to represent nearly optimal catching behavior, considering the time from ball release to impact on palm, which is penalized with the capped distance to the palm center.

One attempt at catching the ball was considered to be one episode, which lasted for 500 timesteps. One simulation step corresponded to 0.01 seconds, giving the system a simulated time of 5 seconds to catch and hold the ball.

Training was stopped after 500 policy updates.

B. Experiment Variations

Several different approaches were compared in this set of experiments, which we will briefly describe here.

1) Gradient Estimators: Most of our experiments were conducted using Williams' REINFORCE gradient estimator. While there are several other policy gradient techniques known in the literature [13], [6], we will only focus on episodic Natural Actor-Critic [14] as a comparison to REINFORCE to demonstrate that SDE can be applied to other estimation methods as well.

2) Setpoints vs. Torques: Learning to control a robot can be done at different levels in the control hierarchy. One possibility is to learn to control the forces, that will be added to each controlled joint of the robot, which we called *Torque Learning*. Depending on the task, it can be easier to learn the angular setpoints for each joint and let an automatic controller add the torques accordingly to quickly reach the setpoints without overshooting too much. We refer to this variation as *Setpoint Learning*.

3) Batch Learning vs. Queued Learning: In *Batch Learning*, we execute a number of episodes (we used 20 episodes throughout the experiments described here) followed by a learning step, where all episodes are taken into account, a gradient is calculated and the parameters of the controller are moved in the direction of the gradient. It is advisable to have a large number of episodes in each batch to gain an accurate estimate of the gradient. However, more episodes also require more time and therefore slow down learning.

What we call *Queued Learning*, is a variation of this scheme that can execute controller updates in very short time intervals. Just like with Batch Learning, we also collect a number of episodes and execute one learning step at the

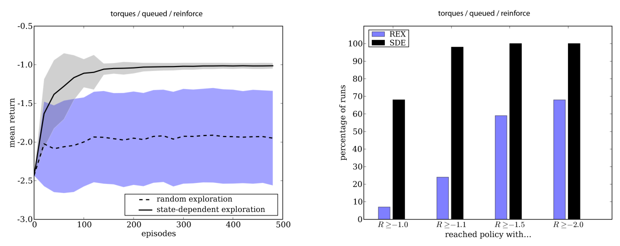


Fig. 3. Results after 100 runs with REINFORCE. Left: The solid and dashed curves show the mean over all runs, the filled envelopes represent the standard deviation. While SDE (solid line) managed to learn to catch the ball quickly in every single case, REX occasionally found a good solution but in most cases did not learn to catch the ball. Right: Cumulative number of runs (out of 100) that achieved a certain level. $R \geq -1$ means "good catch", $R \geq -1.1$ corresponds to all "catches" (closing the hand and holding the ball). $R \geq -1.5$ describes all policies managing to keep the ball on the hand throughout the episode. $R \geq -2$ results from policies that at least slowed down ball contact to the ground. The remaining policies dropped the ball right away.

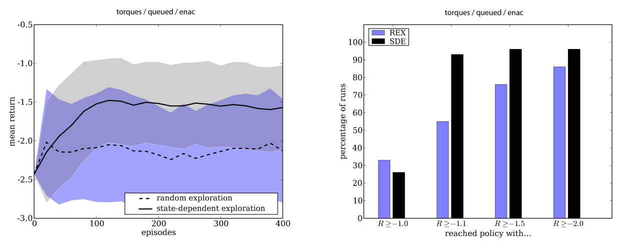


Fig. 4. Results after 100 runs with ENAC. Both learning curves had relatively high variances. While REX often didn't find a good solution, SDE found a catching behavior in almost every case, but many times lost it again due to continued exploration. REX also found slightly more "good catches" but fell far behind SDE considering both "good" and "average" catches.

beginning. Then, however, we discard only the oldest episode from the batch, and add one newly executed episode to the front of the queue, followed by another update step of the current batch, and so forth. This way, we can update our policy after each episode.

C. Results

The best results were achieved when using SDE with REINFORCE gradient estimation, optimal baseline and a learning rate of $\alpha = 0.0001$. The experiment ran with the Queued Learning scheme and learned to control torques directly.

The whole experiment was repeated 100 times. The left side of Figure 3 shows the learning curves over 500 episodes. Please note that the curves are not perfectly smooth because we only evaluated every twentieth policy. As can be seen, SDE finds a near-perfect solution in almost every case, resulting in a very low variance. The mean of the REX experiments indicate a semi-optimal solution, but in fact some of the runs found a good solution while others failed, which explains the high variance throughout the learning process.

The best controller found by SDE yielded a return of -0.95, REX reached -0.97. While these values do not differ much, the chances of producing a good controller are much higher with SDE. The right plot in Figure 3 shows the percentage of runs where a solution was found that was better than a certain value. Out of 100 runs, REX only found a mere 7 policies that qualified as "good catches", where SDE found 68. Almost all SDE runs, 98%, produced rewards $R \geq -1.1$, corresponding to behavior that would be considered a "catch" (closing the hand and holding the ball), although not all policies were as precise and quick as the "good catches". A typical behavior that returns $R \simeq -1.5$ can be described as one that keeps the ball on the fingers throughout the episode but hasn't learned to close the hand. $R \simeq -2.0$ corresponds to a behavior where the hand is held open and the ball falls onto the palm, rolls over the fingers and is then dropped to the ground. Some of the REX trials weren't even able to reach the -2.0 mark. A typical worst-case behavior is pulling back the hand and letting the ball drop to the ground immediately.

To investigate if SDE can be used with different gradient estimation techniques, we ran the same experiments with ENAC [14] instead of REINFORCE. We used a learning rate of 0.01 here, which lead to similar convergence speed. The results are presented in Figure 4. The difference compared to the results with REINFORCE is, that both algorithms, REX and SDE had a relatively high variance. While REX still had problems to converge to stable catches (yet showed a 26% improvement over the REINFORCE version of REX for "good catches"), SDE in most cases (93%) found a "catching" solution but often lost the policy again due to continued exploration, which explains its high variance. Perhaps this could have been prevented by using tricks like reducing the learning rate over time or including a momentum term in the gradient descent. These advancements, however, are beyond the scope of this paper. SDE also had trouble reaching near-optimal solutions with $R \geq -1.0$ and even fell a little behind REX. But when considering policies with $R \geq -1.1$, SDE outperformed REX by over 38%. Overall the experiments show that SDE can in fact improve more advanced gradient estimation techniques like ENAC.

Then we ran the same experiment as stated above, with the same initial parameters and REINFORCE gradient estimation, but learned the controller setpoints for joint angles rather than the forces directly (see section IV-B.2). We used a basic PD controller following this equation $MV(t) = K_p e(t) + K_d \frac{\partial e}{\partial t}$, with $MV(t)$ being the manipulated variable over time, $e(t)$ the difference of the process variable and the setpoint at time t, $K_p = 2.0$ the proportional gain, and $K_d = 0.1$ the derivative gain (the gain constants are tuning parameters that need to be determined experimentally). We expected a performance improvement assuming that it is easier to only learn the desired angles and let a controller take care of the forces, rather than learning the torques directly. As can be seen in Figure 5, lefthand side, this turned out to be the case for returns of $R \geq -1.5$, i.e. keeping the ball on the hand. Now, not only SDE (which was able to reach this performance in all runs even when controlling the torques directly) but also REX achieved at least $R = -1.5$ in 100% of their runs. However, if we look at good catches

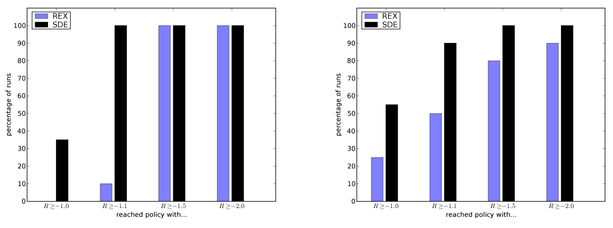


Fig. 5. Cumulative number of runs (as percentages) that achieved a certain return value. Left side: Agent learned setpoints for joint angles, which improves average performance but makes it harder to get results with $R \geq -1.0$ for both REX and SDE. Right side: Agent learned joint torques directly, but in a batch fashion, rather than queued. Convergence speed dropped, overall performance increased slightly for REX and decreased slightly for SDE.

of $R = -1.0$ or better, it turns out that this was more difficult in the setpoint learning experiment for both SDE and REX. Two reasons could explain this fact: firstly, the PD controller gains might not be perfectly tuned, overshooting for small, precise movements and resulting in a somewhat clumsy hand coordination. Secondly, controlling the torques might be less sensitive to noise because the forces add up over time and gain momentum, which smudges and averages the noisy signals over time. This issue will be investigated in a future publication.

Lastly, we compared the performance between batch and queued learning, as described in section IV-B.3. The setup was again identical to the first experiment but used the traditional batch update, discarded the whole batch afterwards and executed a new batch. Because queued learning uses each sample not just once but 20 times (the size of the batch), we expected to be able to increase the learning rate of the batch learning experiment by a factor of 20. However we discovered instabilities with increased learning rate, which lead to big jumps in parameter space, potentially ruining otherwise promising solutions. Therefore we used the same learning rate as with the queued experiment, which lead to a much slower convergence with a factor of 0.09 compared to the queued version. Besides this difference in convergence speed, the overall result did not change dramatically. Looking at the percentages of good catches reached (see Figure 5, on the right), REX performed 18% better with batch learning, while SDE decreased by 12%. Overall, it seems that SDE favours queued learning, while REX performs better with the traditional batch learning approach. It remains to be seen if this holds for different tasks and on real robots as well.

V. Conclusion

We evaluated state-dependent exploration as an alternative to random exploration for policy gradient methods. By creating strategy variations similar to those of finite differences but without their disadvantages, SDE inserts considerably less variance into each rollout or episode. In a robotics simulation task, we investigated in several different learning setups, where SDE could always clearly outperform REX. We found that learning setpoints for automatic control can improve average performance but makes it harder to find excellent solutions. SDE also improves upon recent gradient estimation techniques such as ENAC. Furthermore, SDE is simple and elegant, and easy to integrate into existing policy gradient implementations. All of this recommends SDE as a valuable addition to the existing collection of policy gradient methods. The physics-based ball catching simulation gives a first hint of SDE's performance in real-world applications, while ongoing work is focusing on grasping tasks in realistic robot domains.

VI. Acknowledgements

This work was funded within the Excellence Cluster *Cognition for Technical Systems* (CoTeSys) by the German Research Foundation (DFG).

References

[1] C. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.
[2] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.
[3] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: a survey," *Journal of AI research*, vol. 4, pp. 237–285, 1996.
[4] M. A. Wiering, "Explorations in efficient reinforcement learning," Ph.D. dissertation, University of Amsterdam / IDSIA, February 1999.
[5] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, 1992.
[6] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems*, 2000.
[7] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.
[8] Y. Nakamura, T. Mori, M. Sato, and S. Ishii, "Reinforcement learning for a biped robot based on a CPG-actor-critic method," *Neural Networks*, vol. 20, no. 6, pp. 723–735, 2007.
[9] N. Mitsunaga, C. Smith, T. Kanda, H. Ishiguro, and N. Hagita, "Robot behavior adaptation for human-robot interaction based on policy gradient reinforcement learning," *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pp. 218–225, 2005.
[10] J. Moody and M. Saffell, "Learning to trade via direct reinforcement," *Neural Networks, IEEE Transactions on*, vol. 12, no. 4, pp. 875–889, 2001.
[11] L. Peshkin and V. Savova, "Reinforcement learning for adaptive routing," *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, vol. 2, 2002.
[12] T. Rückstieß, M. Felder, and J. Schmidhuber, "State-dependent exploration for policy gradient methods," in *Proceedings of the Nineteenth European Conference on Machine Learning ECML*, 2008 (in print).
[13] J. Baxter and P. Bartlett, "Reinforcement learning in POMDP's via direct gradient ascent," *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 41–48, 2000.
[14] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," in *Proceedings of the Sixteenth European Conference on Machine Learning*, 2005.
[15] J. Spall, "Implementation of the simultaneous perturbation algorithm forstochastic optimization," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 34, no. 3, pp. 817–823, 1998.
[16] F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber, "Policy gradients with parameter-based exploration for control," in *Proceedings of the International Conference on Artificial Neural Networks ICANN*, 2008 (in print).
[17] A. Ng and M. Jordan, "PEGASUS: A policy search method for large MDPs and POMDPs," *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pp. 406–415, 2000.
[18] D. Aberdeen, *Policy-gradient Algorithms for Partially Observable Markov Decision Processes*. Australian National University, 2003.