PALADYN Journal of Behavioral Robotics

VERSITA

Research Article · DOI: 10.2478/pjbr-2013-0003 · JBR · 2013 · 4(1) · 49-61

Robot Skill Learning: From Reinforcement Learning to Evolution Strategies

Freek Stulp[1,2*], **Olivier Sigaud**[3†]

1 Robotics and Computer Vision
ENSTA-ParisTech
Paris, France

2 FLOWERS Team
INRIA Bordeaux Sud-Ouest
Talence, France

3 Institut des Systèmes Intelligents et de Robotique, Université Pierre Marie Curie CNRS UMR 7222, Paris

Received 17-05-2013

Accepted 18-07-2013

Abstract

Policy improvement methods seek to optimize the parameters of a policy with respect to a utility function. Owing to current trends involving searching in parameter space (rather than action space) and using reward-weighted averaging (rather than gradient estimation), reinforcement learning algorithms for policy improvement, e.g. PoWER and PI^2, are now able to learn sophisticated high-dimensional robot skills. A side-effect of these trends has been that, over the last 15 years, reinforcement learning (RL) algorithms have become more and more similar to evolution strategies such as (μ_W, λ)–ES and CMA-ES. Evolution strategies treat policy improvement as a black-box optimization problem, and thus do not leverage the problem structure, whereas RL algorithms do.

In this paper, we demonstrate how two straightforward simplifications to the state-of-the-art RL algorithm PI^2 suffice to convert it into the black-box optimization algorithm (μ_W, λ)–ES. Furthermore, we show that (μ_W, λ)–ES empirically outperforms PI^2 on the tasks in [36]. It is striking that PI^2 and (μ_W, λ)–ES share a common core, and that the simpler algorithm converges faster and leads to similar or lower final costs.

We argue that this difference is due to a third trend in robot skill learning: the predominant use of dynamic movement primitives (DMPs). We show how DMPs dramatically simplify the learning problem, and discuss the implications of this for past and future work on policy improvement for robot skill learning.

Keywords

reinforcement learning · black-box optimization · evolution strategies · dynamic movement primitives

1. Introduction

To improve the flexibility and adaptiveness of robots – important requirements for robots operating in human environments – an increasing emphasis is being placed on robots that acquire skills *autonomously* through interactions with the environment [29]. State-of-the-art reinforcement learning (RL) algorithms such as PoWER and PI^2 are now able to learn very complex and high-dimensional robot skills [16, 33]. Over the last 15 years, several trends have facilitated the application of policy improvement to robotic skill learning: more effective and safer exploration (by searching in the parameter space rather than the action space), and more robust and safer parameter updates (by using reward-weighted averaging rather than gradient estimation).

In this article, we give an overview of several policy improvement algorithms, and demonstrate that a side-effect of these trends has been that they have brought these algorithms closer and closer to evolution strategies such as CMA-ES (Section 2). Evolution strategies also use parameter space exploration and reward-weighted averaging, but treat policy improvement as a black-box optimization (BBO) problem, and thus do not leverage the problem structure, whereas RL algorithms do.

Although previously *"[t]here has been limited communication between these research fields"* [37], this article contributes to a growing body of work that investigates, formalizes and empirically compares RL and BBO approaches to policy improvement [5, 8, 9, 12, 25, 28, 31, 37]. In Section 3, we draw a strong bridge between RL and BBO, by showing that only two straightforward simplifications suffice to convert the state-of-the-art PI^2 RL algorithm into the evolutionary strategy (μ_W, λ)–ES, a BBO algorithm which is the backbone of CMA-ES [6].

In our empirical comparison in Section 4, we make a rather surprising observation: (μ_W, λ)–ES – one of the most basic evolution strategies – is able to outperform state-of-the-art policy improvement algorithms such as PI^2 and PoWER with policy representations typically considered in the robotics community. This observation raises a pertinent question: Why are much simpler algorithms that use less information able to outperform algorithms specifically designed for policy improvement?

We argue that part of the answer to this question lies in a further trend in robotic skill learning with policy improvement: the predominant use of dynamic movement primitives (DMPs) [11] as the underlying policy. We highlight that, from the perspective of a learning algorithm, DMPs reduce the dimensionality of the action space by optimizing each dimension independently, and reduce the input state space for any task to 1 dimension.

The aforementioned reductions simplify the learning problem so dramatically, that it immediately raises the key question of this paper: Is the recent success of applying RL algorithms to policy improvement for robot skill learning mainly due to improvements in the algorithms, or to the simplification of the problem by using DMPs as the policy representation? We believe this question to be of fundamental importance to robot skill learning. It sheds a new light on the robot skill learning results over the last years, and opens up several important future research directions, which we discuss in Section 6.

*E-mail: freek.stulp@ensta-paristech.fr
†E-mail: olivier.sigaud@upmc.fr

In summary, the main contributions of this article, which also determine its structure, are the following: Section 2: provide an overview of policy improvement methods, and argue how several algorithmic trends have moved these methods closer and closer to BBO. Section 3: demonstrate that two simplifications suffice to convert the state-of-the-art RL algorithm PI^2 into the black-box evolution strategy (μ_W, λ)-ES. Section 4: show that (μ_W, λ)-ES empirically outperforms PI^2 on the tasks in [36]. Section 5: uncover the role that DMPs have in the rather surprising empirical result above. Section 6: discuss the implications of these results.

2. Background

In RL, a policy π maps states to actions. An optimal policy π^* chooses the action that optimizes the cumulative discounted reward over time. When the state and actions sets of the system are discrete, finding an optimal policy π^* can be cast in the framework of discrete Markov Decision Processes (MDPs) and solved with, for instance, Dynamic Programming (DP) [34]. For problems where the state is continuous, many state approximation techniques exist in the field of Approximate Dynamic Programming (ADP) methods [22]. But when the action space also becomes continuous, the extension of DP or ADP methods results in optimization problems that have proven hard to solve in practice [26].

In such contexts, a policy cannot be represented by enumerating all actions, so parametric policy representations $\pi_{\boldsymbol{\theta}}$ are required, where $\boldsymbol{\theta}$ is a vector of parameters. Thus, finding the optimal policy π^* corresponds to finding optimal policy parameters $\boldsymbol{\theta}^*$. As finding the globally optimal policy parameters $\boldsymbol{\theta}^*$ is generally too expensive in the high-dimensional spaces typical for robot skill learning, policy improvement methods are local methods, i.e. they will converge to the next *local* minimum. Choosing an initial parameter vector $\boldsymbol{\theta}^{\text{init}}$ that lies in the vicinity of $\boldsymbol{\theta}^*$ may improve convergence towards $\boldsymbol{\theta}^*$; in robotics $\boldsymbol{\theta}^{\text{init}}$ is usually acquired by bootstrapping through imitation learning [10, 33].

In episodic RL, on which this article focusses, the learner executes a task until a terminal state is reached. Executing a policy from an initial state until the terminal state, called a "Monte Carlo roll-out", leads to a trajectory $\boldsymbol{\tau}$, which may contain information about the states visited, actions executed, and rewards received. Many policy improvement methods use an iterative process of exploration (where the policy is executed K times leading to a set of trajectories $\boldsymbol{\tau}_{k=1\ldots K}$) and parameter updating (where the policy parameters $\boldsymbol{\theta}$ are updated based on $\boldsymbol{\tau}_{k=1\ldots K}$). This process is explained in more detail in the generic policy improvement loop in Figure 1.

In this section, we give an overview of algorithms that implement this generic loop. We distinguish between three main classes of algorithms. The first class is based on putting lower bounds on the expected return (Section 2.1). The second are path integral solutions to stochastic optimal control (Section 2.2). The third class treats policy improvement as a black-box optimization (BBO) problem (Section 2.3).

We highlight in particular whether these algorithms: 1) Perform action *or* parameter perturbation to foster exploration. 2) Use the scalar episode return *or* the rewards at each time step to perform an update. 3) Are actor-critic methods – which approximate a value function – *or* direct policy search methods – which do not. 4) Perform a parameter update by estimating the (natural) gradient *or* through reward-weighted averaging.

The formulae referenced throughout this section are found in Figure 2.

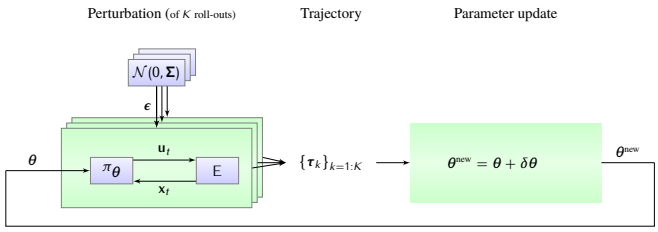

Figure 1. Generic policy improvement loop. In each iteration, the policy is executed K times. One execution of a policy is called a 'Monte Carlo roll-out', or simply 'roll-out'. Because the policy is perturbed (different perturbation methods are described in Section 2.1.3), each execution leads to slightly different trajectories in state/action space, and potentially different rewards. The exploration phase thus leads to a set of different trajectories $\boldsymbol{\tau}_{k=1\ldots K}$. Based on these trajectories, policy improvement methods then update the parameter vector $\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}^{\text{new}}$ such that the policy is expected to incur lower costs/higher rewards. The process then continues with the new $\boldsymbol{\theta}^{\text{new}}$ as the basis for exploration.

2.1. Policy Improvement through Lower Bounds on the Expected Return

We now briefly describe three algorithms that build on one another to achieve ever more powerful policy improvement methods, being REINFORCE [41], eNAC [20], and PoWER [16]. All these algorithms may be derived from a common framework based on the lower bound on the expected return, as demonstrated by Kober and Peters [16]. Since these algorithms have already been covered in extensive surveys [16, 19, 20], we do not present them in full detail here, and focus only on those properties that are relevant to the aims of this article.

An underlying assumption of the algorithms presented in Section 2.1 and 2.2 is that the policies are represented as $\mathbf{u}_t = \psi(\mathbf{x}, t)^\intercal \boldsymbol{\theta}$, where ψ is a set of basis functions, for instance Gaussian kernels, $\boldsymbol{\theta}$ are the policy parameters, $\mathbf{x}$ is the state, and t is time since the roll-out started.

2.1.1. REINFORCE

The REINFORCE algorithm [41] ("reward increment = nonnegative factor × offset reinforcement × characteristic eligibility") uses a stochastic policy to foster exploration: $\mathbf{u}_t = \pi_{\boldsymbol{\theta}}(\mathbf{x}) + \boldsymbol{\epsilon}_t$. Here, $\pi_{\boldsymbol{\theta}}(\mathbf{x})$ returns the nominal motor command[1], and $\boldsymbol{\epsilon}_t$ is a perturbation of this command at time t. In REINFORCE, this policy is executed K times with the same $\boldsymbol{\theta}$, and the states/actions/rewards that result from a roll-out are stored in a trajectory.

Given K such trajectories, the parameters $\boldsymbol{\theta}$ are then updated by first estimating the gradient $\hat{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ (1) of the expected return $J(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{i=1}^{N} r_i | \pi_{\boldsymbol{\theta}}\right]$. Here, the trajectories are assumed to be of equal length, i.e. having N discrete time steps $t_{i=1\ldots N}$. The notation in Equation (1) (to the upper right in Figure 2) estimates the gradient $\hat{\nabla}_{\theta_d} J(\boldsymbol{\theta})$ for each parameter entry d in the vector $\boldsymbol{\theta}$ separately. Riedmiller et al. [23] provide a concise and lucid explanation how to derive (1). The

[1] With this notation, the policy $\pi_{\boldsymbol{\theta}}(\mathbf{x})$ is actually deterministic. A truly stochastic policy is denoted as $\mathbf{u}_t \sim \pi_{\boldsymbol{\theta}}(\mathbf{u}|\mathbf{x}) = \mu(\mathbf{x}) + \boldsymbol{\epsilon}_t$ [23], where $\mu(\mathbf{x})$ is a deterministic policy that returns the nominal command. We use our notation for consistency with parameter perturbation, introduced in Section 2.1.3. For now, it is best to consider the sum $\pi_{\boldsymbol{\theta}}(\mathbf{x}) + \boldsymbol{\epsilon}_t$ to be the stochastic policy, rather than just $\pi_{\boldsymbol{\theta}}(\mathbf{x})$.

baseline (2) is chosen so that it minimizes the variation in the gradient estimate [21]. Finally, the parameters are updated through steepest gradient ascent (3), where the open parameter α is a learning rate.

2.1.2. eNAC

One issue with REINFORCE is that the 'naive', or 'vanilla'[2], gradient $\nabla_\theta J(\theta)$ it uses is sensitive to different scales in parameters. To find the true direction of steepest descent towards the optimum, independent of the parameter scaling, eNAC ("Episodic Natural Actor Critic") uses the Fischer information matrix F to determine the 'natural gradient': $\theta^{new} = \theta + \alpha F^{-1}(\theta)\nabla_\theta J(\theta)$. In practice, the Fischer information matrix need not be computed explicitly [20].

Another difference with REINFORCE is that eNAC uses a value function V_{π_θ} as a compact representation of long-term reward. In continuous state-action spaces, V_{π_θ} cannot be represented exactly, but must be estimated from data. Actor-critic methods therefore update the parameters in two steps: 1) approximate the value function from the point-wise estimates of the cumulative rewards in the roll-out trajectories $\boldsymbol{\tau}_{k=1\ldots K}$; 2) update the parameters using the value function. In contrast, direct policy search methods such as REINFORCE update the parameters directly using the point-wise estimates[3]. The main advantage of having a value function is that it generalizes; whereas K roll-outs provide only K point-wise estimates of the cumulative reward, a value function approximated from these K point-wise estimates is also able to provide estimates not observed in the roll-outs.

Summary from REINFORCE to eNAC. Going from REINFORCE to eNAC represents a transition from vanilla to natural gradients, and from direct policy search to actor-critic.

2.1.3. PoWER

REINFORCE and eNAC are both 'action perturbing' methods which perturb the nominal command at each time step $\mathbf{u}_t = \mathbf{u}_t^{\text{nominal}} + \boldsymbol{\epsilon}_t$. Action-perturbing algorithms have several disadvantages: 1) Samples are drawn independently from one another at each time step, which leads to a very noisy trajectory in action space [25]. 2) Consecutive perturbations may cancel each other and are thus washed out [16]. The system also often acts as a low-pass filter, which further reduces the effects of perturbations that change with a high frequency. 3) On robots, high-frequency changes in actions, for instance when actions represent motor torques, may lead to dangerous behavior, or damage to the robot [25]. 4) It causes a large variance in parameter updates, an effect which grows with the number of time steps [16].

The "Policy Learning by Weighting Exploration with the Returns" (PoWER) algorithm therefore implements a different policy perturbation scheme first proposed by Rückstiess et al. [24], where the parameters θ of the policy, rather than its output[4], are perturbed, i.e. $\pi_{[\theta+\epsilon_t]}(\mathbf{x})$ rather than $\pi_\theta(\mathbf{x}) + \boldsymbol{\epsilon}_t$. This distinction has been illustrated in Figure 2 (upper left charts).

REINFORCE and eNAC estimate gradients, which is not robust when noisy, discontinuous utility functions are involved. Furthermore, they require the manual tuning of the learning rate α, which is not straightforward, but critical to the performance of the algorithms [16, 36]. The PoWER algorithm proposed by Kober and Peters [16] addresses these issues by using reward-weighted averaging, which rather takes a weighted average of a set of K exploration vectors $\boldsymbol{\epsilon}_{k=1\ldots K}$ as in (8) in Figure 2. Here, K refers to the number of roll-outs, and ψ_{t_i} is a vector of the basis function activations at time t_i. The computation of $\delta\theta$ (8) may be interpreted as taking the average of the perturbation vectors $\boldsymbol{\epsilon}_k$, but weighting each with $S_t^k / \sum_{l=1}^K S_t^l$, which is a normalized version of the reward-to-go S_t^k. Hence the name reward-weighted averaging. An important property of reward-weighted averaging is that it follows the natural gradient [1], *without* having to actually compute the gradient or the Fischer information matrix.

The final main difference between REINFORCE/eNAC and PoWER is the information contained in the trajectories $\boldsymbol{\tau}_{k=1:K}$ resulting from the roll-outs. For REINFORCE/eNAC, trajectories must contain information about the state and actions to compute $\nabla_\theta \log \pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$ in the update. In PoWER the policy parameter perturbations $\boldsymbol{\epsilon}_t$ are stored rather than the output of the perturbed policy $\mathbf{u}_t$. And the activations of the basis functions ψ_t are stored rather than the states $\mathbf{x}_t$[5]. Finally, the rewards at each time step r_t are required to compute the cost-to-go (6).

Summary from eNAC to PoWER. Going from eNAC to PoWER represents a transition from action perturbation to policy parameter perturbation, from estimating the gradient to reward-weighted averaging, and from actor-critic back to direct policy search (as in REINFORCE).

2.2. Policy Improvement with Path Integrals

The Policy Improvement with Path Integrals (PI^2) algorithm has its roots in path integral solutions of stochastic optimal control problems[6]. In particular, PI^2 is the application of Generalized Path Integral Control (GPIC) to parameterized policies [36]. In this article, we provide a post-hoc interpretation of PI^2's update rule in Figure 2; for the complete PI^2 derivation we refer to Theodorou et al. [36].

Given the roll-outs, let us now explain how PI^2 updates the parameters (right block in Figure 2 labeled "PI^2"). First, the cost-to-go $S_{i,k}$ is computed for each of the K roll-outs and each time step $i = 1 \ldots N$. The terminal cost ϕ_N and immediate costs r_i are task-dependent and provided by the user[7]. $\mathbf{M}_{j,k}$ is a projection matrix onto the range space of ψ_j under the metric $\mathbf{R}^{-1}$, cf. [36]. The probability of a roll-out $P_{i,k}$ is computed as the normalized exponentiation of the cost-to-go. This assigns high probabilities to low-cost roll-outs and vice versa. The intuition behind this step is that trajectories of lower cost should have

[2] 'Vanilla' refers to the canonical version of an entity. The origin of this expression lies in ice cream flavors; i.e. 'plain vanilla' vs. 'non-plain' flavors, such as strawberry, chocolate, etc.

[3] The point-wise estimates are sometimes considered to be a special type of critic; in this article we use the term 'critic' only when it is a function approximator.

[4] In the PoWER derivation, the policy $\pi(\mathbf{u}|\mathbf{x}, t)$ is defined as $\mathbf{u} = \theta^\mathsf{T}\psi(\mathbf{x}, t) + \epsilon(\psi(\mathbf{x}, t))$. By choosing the state-dependent exploration as $\epsilon(\psi(\mathbf{x}, t)) = \boldsymbol{\epsilon}_t^\mathsf{T}\psi(\mathbf{x}, t)$, and plugging it into the policy definition, we acquire $\mathbf{u} = \theta^\mathsf{T}\psi(\mathbf{x}, t) + \boldsymbol{\epsilon}_t^\mathsf{T}\psi(\mathbf{x}, t)$. See also equations on page 8 and 9 of [16]. When rewriting this as $\mathbf{u} = (\theta + \boldsymbol{\epsilon}_t)^\mathsf{T}\psi(\mathbf{x}, t)$, we see that the policy parameters θ are perturbed directly. This form of the perturbed policy is most clearly described in Appendix A.3 of [16].

[5] In the original formulation (Algorithm 4 of [16]), $\mathbf{x}$ and t are also stored, and then used to compute $\psi(\mathbf{x}, t)$ in the update. Since the parameters θ do not depend on the state, and the sum in (8) is only over t (i.e. not over $\mathbf{x}$), we may store only ψ_t instead. The only difference is whether $\psi(\mathbf{x}, t)$ is computed during the roll-out (in which case we need $\mathbf{x}$ in the trajectory), or afterwards (in which case we do not). The same holds for $r(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1}, t)$, which may also be stored as r_t.

[6] Note that in the previous section, algorithms aimed at maximizing *rewards*. In optimal control, the convention is rather to define *costs*, which should be minimized.

[7] In PI^2, r_t is split into immediate costs q_i and command costs $\frac{1}{2}\theta_i^T\mathbf{R}\theta_i$, with command cost matrix $\mathbf{R}$.

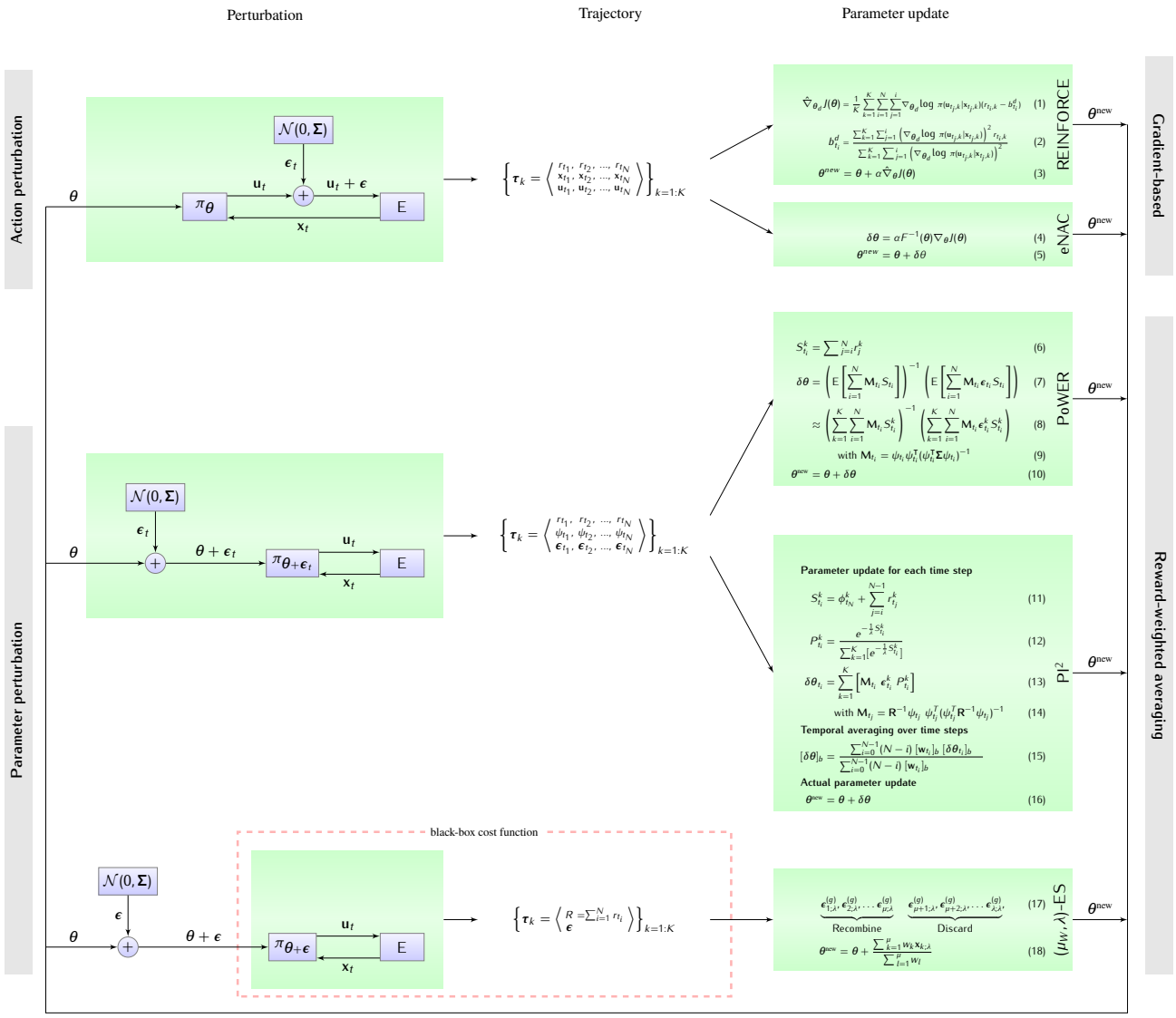

Figure 2. Illustrations of different policy improvement algorithms. The right (green) blocks depict the exploration method (action or parameter perturbing). The center column depicts the information stored in the trajectories. This information is required to perform the parameter update. The right column shows how the different algorithms update the parameters.

higher probabilities. The interpretation of $P_{i,k}$ as a probability follows from applying the Feynman-Kac theorem to the SOC problem, cf. [36]. The key algorithmic step is in (13), where the parameter update $\delta\boldsymbol{\theta}$ is computed for each time step i through weighted averaging over the exploration $\boldsymbol{\epsilon}$ of all K trials. Trajectories with higher probability $P_{i,k}$ (due to a lower cost $S_{i,k}$) therefore contribute more to the parameter update.

A different parameter update $\delta\boldsymbol{\theta}_i$ is computed for each time step. To acquire one parameter vector $\boldsymbol{\theta}$, the time-dependent updates must be averaged over time, one might simply use the mean parameter vector over all time steps: $\delta\boldsymbol{\theta} = \frac{1}{N}\sum_{i=1}^{N}\delta\boldsymbol{\theta}_i$. Although such temporal averaging is necessary, the particular weighting scheme used in temporal averaging does not follow from the derivation. Rather than a simple mean, Theodorou et al. [36] suggest the weighting scheme in Eq. (15). It emphasizes updates earlier in the trajectory, and also makes use of the activation of the j^{th} basis function at time step i, i.e. $w_{j,i}$ (32).

As demonstrated in [36], PI^2 is able to outperform the previous RL algorithms for parameterized policy learning described in Section 2.1 by at least one order of magnitude in learning speed (number of roll-outs to converge) and also lower final cost performance. As an additional benefit, PI^2 has no open algorithmic parameters, except for the magnitude of the exploration noise $\boldsymbol{\Sigma}$, and the number of trials per update K.

Although having been derived from very different principles, PI^2 and PoWER use identical policy perturbation methods, and have very similar update rules, i.e. compare the piecewise similarity of equations

(6)/(11), (8)/(13), and (9)/(14) in Figure 2. The shared core step of reward-weighted averaging is apparent in $\sum_{k=1}^{K} \mathbf{M}_{t_i} \boldsymbol{\epsilon}_{t_i} S_{t_i}$ (8) and $\sum_{k=1}^{K} \mathbf{M}_{t_i} \boldsymbol{\epsilon}_{t_i}^k P_{t_i}^k$ (13). The main difference is that PoWER requires the cost function to behave as improper probability, i.e. be strictly positive and integrate to a constant number. This constraint can make the design of suitable cost functions more complicated [36]. For cost functions with the same optimum in PoWER's pseudo-probability formulation and PI^2 cost function, "both algorithms perform essentially identical" [36]. Another difference is that matrix inversion in PoWER's parameter update (8) is not necessary in PI^2.

Summary comparison of PoWER and PI^2. PI^2 and PoWER use identical policy perturbation methods and very similar update rules. The main difference is that PI^2 allows for more flexible cost function design.

2.3. Policy Improvement as Black-Box Optimization

As is pointed out by Rückstiess et al. [25], *"the return of a whole RL episode can be interpreted as a single fitness evaluation. In this case, parameter-based exploration in RL is equivalent to black-box optimization."*. This constitutes a qualitatively different approach to policy improvement, because it considers only the scalar return of an episode, and does not take into account the state/action pairs visited, or the rewards received at each time step. Considering policy improvement as a black-box optimization (BBO) problem allows generic BBO algorithms, such as evolution strategies, to be applied to policy improvement.

We now present the basic formalization underlying BBO and evolution strategies, and then describe evolution strategies that have been applied to policy improvement.

2.3.1. Evolution Strategies

Optimization aims at finding the best solution x^* that optimizes an objective function J, i.e. $x^* = \operatorname{argmax} J(x)$. In *continuous* optimization, X is a continuous solution space in $\mathbb{R}^d$. In *black-box* optimization, J is unknown – a black-box – and the optimum can thus not be determined analytically.

Evolution strategies (ES) are algorithms for solving continuous BBO problems. They belong to the class of evolutionary algorithms, which use mutation, recombination, and selection to iteratively acquire increasingly good (populations of) candidate solutions [3].

One of the most basic ES algorithms is the standard (μ,λ)-ES, with global intermediate recombination[8] in object parameter space X [3, 27], listed in (19)-(21). This algorithm first generates λ mutated offspring[9] from the parent parameter vector $\mathbf{x}^{(g)}$, by sampling λ perturbation vectors $\boldsymbol{\epsilon}_k^{(g)}$ from a Gaussian distribution (19). It then updates the parent through recombination, by taking the mean of the μ offspring with the best utilities $J(\mathbf{x}_k)$. In (21), $\mathbf{x}_{k;\lambda}$ denotes the k-th best offspring individual. The updated parameter $\mathbf{x}$ then serves as the parent for the next generation.

$$\forall k = 1 \ldots \lambda : \boldsymbol{\epsilon}_k^{(g)} = \sigma \mathcal{N}(0, \mathbf{I}) \qquad \text{Mutate} \qquad (19)$$

$$\underbrace{\boldsymbol{\epsilon}_{1;\lambda}^{(g)}, \ldots \boldsymbol{\epsilon}_{\mu;\lambda}^{(g)}}_{\text{Low cost: Recombine}} \quad , \quad \underbrace{\boldsymbol{\epsilon}_{\mu+1;\lambda}^{(g)}, \ldots \boldsymbol{\epsilon}_{\lambda;\lambda}^{(g)}}_{\text{High cost: Discard}} \qquad \text{Select} \qquad (20)$$

$$\mathbf{x}^{(g+1)} = \mathbf{x}^{(g)} + \frac{1}{\mu} \sum_{k=1}^{\mu} \boldsymbol{\epsilon}_{k;\lambda}^{(g)} \qquad \text{Recombine} \qquad (21)$$

A variation on this algorithm is (μ_W,λ)-ES [6], where the recombination is a *weighted* average, with weight vector $w_{i\ldots\mu}$, cf. (24). Thus, (μ,λ)-ES is a special case of (μ_W,λ)-ES where the weights are all $\frac{1}{\mu}$. To make this relation explicit, (μ,λ)-ES is sometimes written as (μ_I,λ)-ES. Another variation is to sample from a multi-variate Gaussian distribution with covariance matrix $\sigma\mathcal{N}(0,\boldsymbol{\Sigma})$ [6], rather than sampling using a scalar matrix $\sigma\mathcal{N}(0,1) \equiv \mathcal{N}(0,\sigma^2\mathbf{I})$ as in (19). These two variations lead to what we refer to as (μ_W,λ)-ES in the rest of this article; it is visualized in the bottom row of Figure 2, and repeated in (22)-(24).

$$\forall k = 1 \ldots \lambda : \boldsymbol{\epsilon}_k^{(g)} = \sigma \mathcal{N}(0, \boldsymbol{\Sigma}) \qquad \text{Mutate} \qquad (22)$$

$$\underbrace{\boldsymbol{\epsilon}_{1;\lambda}^{(g)}, \ldots \boldsymbol{\epsilon}_{\mu;\lambda}^{(g)}}_{\text{Low cost: Recombine}} \quad , \quad \underbrace{\boldsymbol{\epsilon}_{\mu+1;\lambda}^{(g)}, \ldots \boldsymbol{\epsilon}_{\lambda;\lambda}^{(g)}}_{\text{High cost: Discard}} \qquad \text{Select} \qquad (23)$$

$$\mathbf{x}^{(g+1)} = \mathbf{x}^{(g)} + \frac{\sum_{k=1}^{\mu} w_k \mathbf{x}_{k;\lambda}}{\sum_{l=1}^{\mu} w_l} \qquad \text{Recombine} \qquad (24)$$

These rather simple algorithms form the core of state-of-the-art optimization algorithms such as the Cross-Entropy Method (CEM) with Gaussian distributions [10] and Covariance Matrix Adaptation - Evolutionary Strategy (CMA-ES, the full name is (μ_W,λ)-CMA-ES)[11].

2.3.2. Policy Improvement with Evolution Strategies

Because ES algorithms treat the objective function as a black-box function, they can readily be applied to policy improvement, by 1) interpreting the inputs $\mathbf{x}$ to the objective function as policy parameter vectors, which we denote $\boldsymbol{\theta}$, and 2) having the objective function return a scalar that is the return of an episode, i.e. the sum of the rewards $R = \sum_{i=1}^{N} r_i$.

With this translation, "generate perturbation" in Figure 1 thus corresponds to "mutation" in (19), and the "parameter update" to "recombination" in (21). From the point of view of the optimization algorithm, the roll-out of the policy is completely encapsulated within the black-box objective function; the algorithm is agnostic about whether it is optimizing a set of policy parameters for a robotic task, or the function $J(\boldsymbol{\theta}) = \boldsymbol{\theta}^2$.

Examples of BBO algorithms that have been applied to policy improvement include CEM [4, 17], CMA-ES [8, 25], PGPE [28] and Natural Evolution Strategies (NES) [40].

2.4. Summary

REINFORCE is a direct policy search method that uses action perturbation and gradient-based parameter updates. A main advantage of eNAC over REINFORCE is that it uses the *natural* gradient, rather than the vanilla gradient. PoWER, although sharing much of its derivation with REINFORCE and eNAC, introduces three main adaptations to eNAC: 1) it perturbs the policy parameters rather than the actions,

[8] The complete notation would be $(\mu/\rho,\lambda)$-ES, but as in [6], we set $\rho = \mu$ and drop ρ.

[9] In the RL algorithms of previous sections, the capital letter K is used instead of the symbol λ.

[10] Without covariance matrix adaptation, i.e. by dropping (34) [1] from CEM, it is equivalent to (μ_I,λ)-ES.

[11] With the algorithmic parameters $\mathbf{C}^{(0)} = \mathbf{I}$ and $c_\mu = 0$ in (15) and $c_\sigma = 0$ in (34) in [7], step-size adaptation and covariance matrix adaptation are switched off, and in this case CMA-ES reduces to (μ_W,λ)-ES.

which leads to safer, more informative exploration. 2) it is based on reward-weighted averaging rather than gradient estimation, which leads to more robust parameter updates. 3) it is completely model-free, and does not require policy derivatives. Figure 3 visualizes the chronology of this derivation, and classifies the algorithms according to several algorithmic properties.

Although PI^2 is derived from very different principles and based on Generalized Path Integral Control (GPIC), it is very similar to PoWER. They were introduced nearly simultaneously, and share the three advantages above. Additionally, PI^2 places less constraints on cost functions, facilitating their design in practice.

If the return of an episode is interpreted as a single fitness evaluation, policy improvement may be considered a BBO problem [25]. This allows policy improvement to be solved by BBO algorithms, for example evolution strategies such as (μ_W, λ)-ES, CEM, or CMA-ES.

Figure 3 highlights one of the main sources of inspiration for this work: policy improvement algorithms based on reinforcement learning have become more and more similar to evolution strategies such as CEM and CMA-ES. This raises several questions: 1) What exactly is the characteristic difference between BBO and RL, and which modifications are necessary to convert an RL algorithm for policy improvement, such as PI^2, into a BBO algorithm, such as (μ_W, λ)-ES? 2) How do BBO variants of RL algorithms compare in terms of performance? We address these questions in Section 3 and 4 respectively.

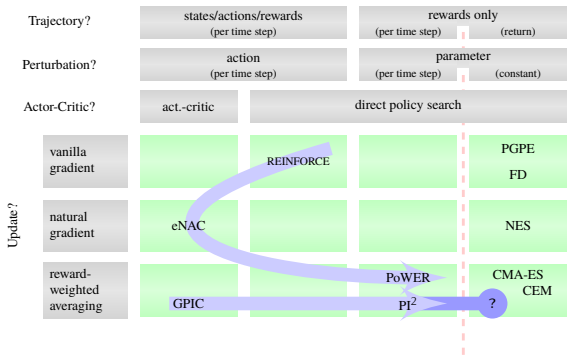

Figure 3. Classification of policy improvement algorithms, given a selection of their properties as discussed throughout Section 2.

3. From PI^2 to (μ_W, λ)-ES

Figure 3 suggests that two modifications are required to convert PI^2 into a BBO algorithm. First of all, the policy perturbation method must be adapted. PI^2 and BBO both use policy parameter perturbation, but in PI^2 it varies over time (i.e. $\boldsymbol{\theta} + \boldsymbol{\epsilon}_t$), whereas it must remain constant over time in BBO (i.e. $\boldsymbol{\theta} + \boldsymbol{\epsilon}$). In Section 3.1, we therefore simplify the perturbation method in PI^2 to be constant over time, which yields the algorithm variation PI^{2-}. Second, we must adapt PI^2 such that it is able to update the parameters based only on the scalar episode return $R = \sum_{i=1}^{N} r_{t_i}$, rather than having access to the reward r_t at each time step t. This is done in Section 3.2.

3.1. Simplifying the Exploration Method

In this section, we present three previously proposed alternative exploration methods [31, 35, 36] for PI^2. In the canonical version of PI^2, the policy parameters are perturbed with a different perturbation at each time step, i.e. $\boldsymbol{\epsilon}_t \sim \mathcal{N}(0, \boldsymbol{\Sigma})$. We refer to this version of PI^2 as PI$^{2\wr\!\wr}$, where the small (blue) symbol serves as a mnemonic to indicate that exploration varies at a high frequency.

In practice, this time-varying exploration has several disadvantages, which have been pointed out in Section 2.1.3. In practical applications of PI^2, the noise is therefore not varied at every time step. For instance, Theodorou et al. [36] and Tamosiumaite et al. [35] generate exploration noise only for the basis function with the highest activation. We refer to this second method as PI^2 with exploration per basis function, or PI$^{2\wr\!\wr}$, where the small (green) symbol serves as a mnemonic of the shape of the exploration for one basis function.

Alternatively, $\boldsymbol{\epsilon}_{t_i,k}$ can be set to have a constant value during a roll-out [33]. Or, as Sehnke et al. [28] explain *"parameters are sampled from [the] distribution at the start of each sequence, and thereafter the controller is deterministic."* Thus, for each of the K roll-outs, we generate $\boldsymbol{\epsilon}_k$ exploration vectors before executing the policy, and keep it constant during the execution, i.e. $\boldsymbol{\epsilon}_{t_i,k} = \boldsymbol{\epsilon}_k$. We call this 'PI^2 with constant exploration', and denote it as PI^{2-}, where the horizontal line indicates a constant value over time. Note that $\boldsymbol{\epsilon}_k$ will still have a temporally extended effect, because it is multiplied with a basis function that is active throughout an extended part of the movement.

3.2. Simplifying the Parameter Update

The PI^2 parameter update leads to a new parameter vector $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \delta\boldsymbol{\theta}$. However, a different parameter update is computed for each time step $\delta\boldsymbol{\theta}_{t_i=1\ldots N}$. Therefore, there is a need to condense the N parameters updates $\delta\boldsymbol{\theta}_{t_i=1\ldots N}$ into one update $\delta\boldsymbol{\theta}$. This step is called temporal averaging, and was proposed by Theodorou et al. [36] as:

$$[\delta\boldsymbol{\theta}]_d = \frac{\sum_{i=1}^{N} (N - i + 1)\, w_{d,t_i}\, [\delta\boldsymbol{\theta}_{t_i}]_d}{\sum_{i=1}^{N} w_{d,t_i} (N - i + 1)}. \tag{25}$$

This temporal averaging scheme emphasizes updates earlier in the trajectory, and also makes use of the basis function weights w_{d,t_i}. However, since this does not directly follow from the derivation *"[u]sers may develop other weighting schemes as more suitable to their needs."* [36]. As an alternative, we now choose a weight of 1 at the first time step, and 0 for all others. This means that all updates $\delta\boldsymbol{\theta}_{t_i}$ are discarded, except the first one $\delta\boldsymbol{\theta}_{t_1}$, which is based on the cost-to-go at the first time step $S^k_{t_1}$. By definition, the cost-to-go at t_1 represents the cost of the entire trajectory. This implies that we must only compute the cost-to-go $S^k_{t_i}$ and probability $P^k_{t_i}$ for $i = 1$.

Simplifying temporal averaging in this way depends strongly on using constant exploration noise during a roll-out. If the noise varies at each time step or per basis function, the variation at the first time step $\boldsymbol{\epsilon}_{t_1,k}$ is not at all representative for the variations throughout the rest of the trajectory.

Finally, as demonstrated in [32], the projection matrix $\mathbf{M}$ may be dropped from the the parameter update in (13), which is our second simplification to the update rule.

This simplified PI^2 variant, with constant exploration noise and without temporal averaging, is listed at the center of Figure 4. The lines that are dropped from PI^2 are highlighted with (dark) red banner; the reason for it being dropped is indicated by a label ⓧ (constant exploration over

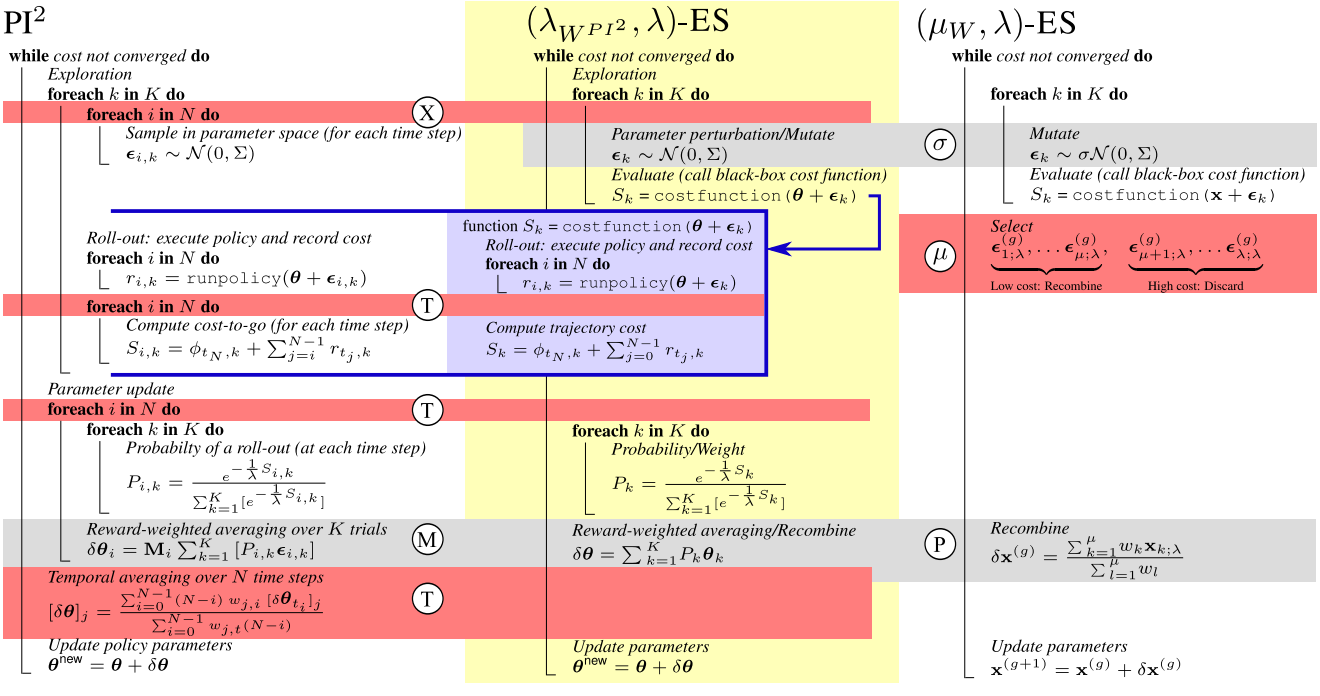

Figure 4. Line by line comparison of PI^2 (left), (μ_W, λ)-ES (right) and $(\lambda_{W^{PI^2}}, \lambda)$-ES (center). Red areas (dark) indicate that a line is dropped from PI^2 or (μ_W, λ)-ES, (light) gray areas indicate a modification. The blue area represents the black-box cost-function used in $(\lambda_{W^{PI^2}}, \lambda)$-ES. This graph illustrates that $(\lambda_{W^{PI^2}}, \lambda)$-ES is a special case of both PI^2 and (μ_W, λ)-ES.

time), (T) (no temporal averaging) and (M) (drop the projection matrix $\mathbf{M}$). Note that we have dropped the subscript $_i$ in this center algorithm, because it is the same $_{i=1}$ for all.

3.3. Simplified PI^2 is a special case of (μ_W, λ)-ES

We now demonstrate that the simplified PI^2 is a special case of the generic (μ_W, λ)-ES algorithm. First, we choose $\mu = \lambda$. This means that all of the samples are used for the parameter update (recombination), and none of the worst are discarded[12]. Then, the weights W in (μ_W, λ)-ES are chosen to be the weights of PI^2, i.e. the weights $w_k = P_k$. Finally, the step-size σ is set to 1, such that exploration is entirely determined by the covariance matrix $\mathbf{\Sigma}$. In Figure 4, these parameter choices have been denoted (μ), (P), and (σ) respectively. We denote this specific case of (μ_W, λ)-ES as $(\lambda_{W^{PI^2}}, \lambda)$-ES, where W^{PI^2} indicates that the weights W are set to the trajectory probabilities as determined by PI^2.

[12] In the footnote on page 919 of [35], Tamosiumaite et al. provide an anecdotal account of testing (μ_W, λ)-ES variants of PI^2, where $\mu \neq \lambda$: *"For example, we did some experiments with PI^2, by introducing a discount factor, as well as using only the best trials of the epoch for weight update [...] Most of these modification did not significantly change learning success."*

Summary from PI^2 and (μ_W, λ)-ES to $(\lambda_{W^{PI^2}}, \lambda)$-ES.

By choosing constant exploration noise and switching off temporal averaging, PI^2 can be reduced to a special case of (μ_W, λ)-ES, with $\mu = \lambda$, $\sigma = 1$, and $w_k = P_k$. $(\lambda_{W^{PI^2}}, \lambda)$-ES is thus a special case of both PI^2 and (μ_W, λ)-ES. We find it very striking that an algorithm that has been derived from stochastic optimal control with solution methods from quantum mechanics [13, 36] shares a common core with an older algorithm from the quite distinct field of evolution strategies.

4. Empirical Comparison

The aim of this section is to empirically compare the different exploration variants of PI^2 (PI$^{2^W}$, PI$^{2^{\Pi}}$, PI$^{2^-}$) as well as PI^2's BBO variant $(\lambda_{W^{PI^2}}, \lambda)$-ES. The comparison is done in terms of convergence speed and final cost of the optimized policy. The experiments are based on the same tasks as presented by Theodorou et al. [36], and described in Appendix A. The main advantage of using these same tasks is that it enables a direct comparison with the results reported by Theodorou et al. [36]. Furthermore, it alleviates us of the need to demonstrate that PI^2 outperforms REINFORCE, eNAC and PoWER on these tasks, because this was already done by Theodorou et al. [36].

For each learning session, we are interested in comparing the convergence speed and final cost, i.e. the value to which the learning curve converges. Convergence speed is measured as the parameter update after which the cost drops below 5% of the initial cost before learning. The final cost is the mean cost over the last 100 updates. For all

tasks and algorithm settings, we execute 10 learning sessions (which together we call an 'experiment'), and report the $\mu \pm \sigma$ over these 10 learning sessions. For all experiments, the DMP and PI^2 parameters are the same as in [36], and listed in Appendix A.

Figure 5 summarizes the results[13] of comparing the four variants of PI^2: the three different exploration methods with temporal averaging (PI$^{2\curlywedge}$, PI$^{2\sqcap}$, PI^{2-}), and PI^2 without temporal averaging, which is equivalent to $(\lambda_{W^{PI^2}}, \lambda)$-ES. The learning curves are the $\mu \pm \sigma$ over 10 learning sessions for Task 4.

To evaluate the convergence speed, we determine when each of the learning curves drops below 5% of the cost before learning, as highlighted in the left graph. The means of these values for the three exploration methods are visualized as vertical lines in the left graph of Figure 5. At the top of these lines, a horizontal bar represents the standard deviation. For convergence we see that $(\lambda_{W^{PI^2}}, \lambda)$-ES <PI^{2-} <PI$^{2\sqcap}$ <PI$^{2\curlywedge}$ (10.4<19.7<26.3<54.9); these differences are significant (p-value < 0.001).

The right graph in Figure 5 compares the final cost of each method, and depicts the average learning curve during the last 100 updates, after which all learning curves have converged. The vertical lines and horizontal bars in the right graphs visualize the $\mu \pm \sigma$ of the final cost over the 10 learning sessions, where the final cost is defined as the mean over a learning curve during the last 100 updates. Again, we see that $(\lambda_{W^{PI^2}}, \lambda)$-ES <PI^{2-} ≈PI$^{2\sqcap}$ <PI$^{2\curlywedge}$ (0.91<1.11≈1.15<1.41); these differences are significant (p-value < 0.001), except for PI^{2-} and PI$^{2\sqcap}$ (p-value $= 0.10$).

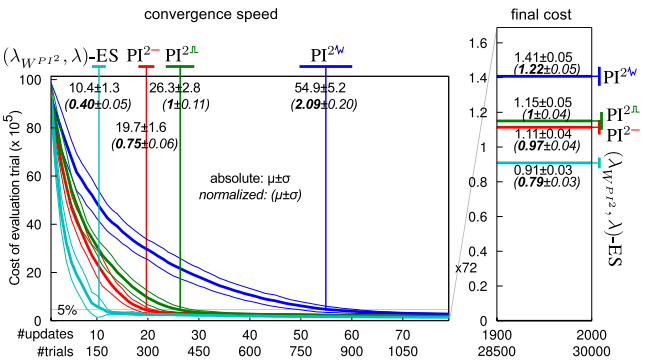

Figure 5. Learning curves of the four PI^2 variants for Task 4, $\mu \pm \sigma$ over 10 learning sessions. The left graph emphasizes convergence speed, the right graph the final cost after costs have converged.

Figure 6 summarizes these results for all five tasks from [36], described in the Appendix A. For each task, all values have been normalized w.r.t. the value for PI$^{2\sqcap}$, because this is the method used by Theodorou et al. [36]. For instance, for Task 4, the convergence below 5% of the initial cost in the bottom graph of Figure 5 is on average at updates 54.9, 26.3, 19.7, 10.4, which, when normalized with the result for PI$^{2\sqcap}$ (26.3), becomes 2.09, 1, 0.75, 0.40 (see bold numbers in Figure 5 and Figure 6). The same normalization is done for the final costs. This normalization makes it easier to compare differences between algorithms, independent of the arbitrary scales in costs between the different tasks.

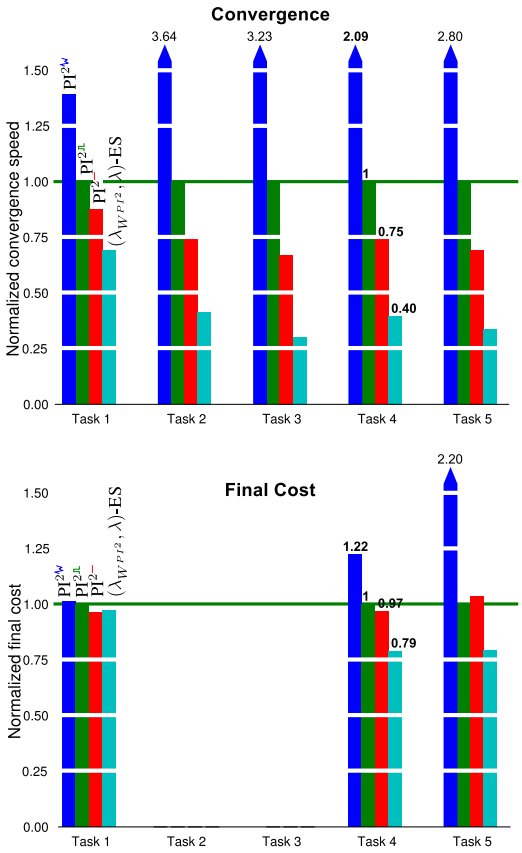

Figure 6. Summary of the normalized convergence speed (top) and normalized final costs (bottom), for all five tasks in Appendix A.

For the convergence speed (top graph), we see the same trend for each task: $(\lambda_{W^{PI^2}}, \lambda)$-ES <PI^{2-} <PI$^{2\sqcap}$ <PI$^{2\curlywedge}$, which is significant (p-value < 0.001) except for Task 1 for PI^{2-} <PI$^{2\sqcap}$ with p-value < 0.030). Over all tasks, $(\lambda_{W^{PI^2}}, \lambda)$-ES on average requires only 43% of the convergence time of PI$^{2\sqcap}$, and 57% of PI^{2-}. For the final cost (bottom graph), the differences between the methods are not as consistent for all tasks. However, $(\lambda_{W^{PI^2}}, \lambda)$-ES leads to significantly lower costs (Task 4&5, p-value < 0.001), not significantly different (Task 1 PI$^{2\sqcap}$/$(\lambda_{W^{PI^2}}, \lambda)$-ES, p-value $= 0.03$), or the same (Task 2&3, all methods lead to a final cost of 0).

Summary: $(\lambda_{W^{PI^2}}, \lambda)$-ES converges significantly (p-value<0.001) and substantially (1.75 = 1/0.57) faster than all exploration variants of PI^2. Furthermore, it leads to lower or same final costs, depending on the task.

[13] The results in Figure 5 may be directly compared with Figure 3a) in [36], which also contains the learning curves for REINFORCE and eNAC. The only algorithmic difference is that, to study convergence towards the final cost, we use an exploration decay factor of $\gamma = 0.99$ (see Appendix A). Theodorou et al. [36] do not apply exploration annealing as it *"really only affect[s] fine tuning of the algorithm"*.

5. How Dynamic Movement Primitives Facilitate Policy Improvement

It is rather surprising that the state-of-the-art reinforcement learning algorithm PI^2 is outperformed by its simpler BBO variant $(\lambda_{W^{PI^2}}, \lambda)$-ES, which happens to be a special case of the older (μ_W, λ)-ES algorithm. In this next section, we argue that DMPs are the main reason for this result, because they simplify the reinforcement learning problem so dramatically, that much simpler algorithms which do not leverage the problem structure are able to achieve similar or even better results. After introducing DMPs in the next subsection, we explain two ways in which DMPs simplify the RL problem.

5.1. Dynamic Movement Primitives (DMPs)

DMPs have become the predominant policy representation for robotic skill learning with policy improvement [10, 16, 35]. The core idea behind DMPs is to perturb a simple linear spring-damper system f_t (27) with a non-linear forcing term $h_{\boldsymbol{\theta}}$ (28), to acquire smooth movements of arbitrary shape.

$$\frac{1}{\tau}\ddot{x}_t = f_t + h_{\boldsymbol{\theta}}(s_t)s_t(g - x_0) \quad \text{Transf. system} \quad (26)$$

$$f_t = \alpha(\beta(g - x_t) - \dot{x}_t) \quad \text{Spring-damper} \quad (27)$$

$$h_{\boldsymbol{\theta}}(s_t) = \text{Chosen by user} \quad \text{Func. approx.} \quad (28)$$

$$\frac{1}{\tau}\dot{s}_t = -\alpha s_t \quad \text{Canon. system} \quad (29)$$

The output of a *transformation system* (26) is the acceleration profile of a reference trajectory. The intuition of this approach is to create desired trajectories $x_t, \dot{x}_t, \ddot{x}_t$ for a motor task out of the time evolution of a nonlinear attractor system, where the goal g is a point attractor and x_0 the start state. The parameter vector $\boldsymbol{\theta}$ determines the shape of the attractor landscape, which allows to represent almost arbitrary smooth trajectories. In policy improvement, $\boldsymbol{\theta}$ are typically the parameters that are optimized with respect to a cost function [36], though more recently other DMP parameters have also been optimized [33, 35].
The canonical system s_t is the phase of the movement, which is 1 at the beginning, and decays to 0 over time. The multiplication of $h_{\boldsymbol{\theta}}(s_t)$ with s_t in (26) ensures that the effect of $h_{\boldsymbol{\theta}}$ disappears towards the end of the movement when $s = 0$. The entire system thus converges to the attractor g, which is known as the goal of the movement.
The non-linear forcing term is a function approximator parameterized with $\boldsymbol{\theta}$. A common approach is to multiply the parameter vector with a set of Gaussian kernels as in (30)-(32). The parameters $\boldsymbol{\theta}$ may be learned from observed trajectories through locally weighted (projection) regression [39].

$$h(s_t) = \psi(s_t)^\intercal \boldsymbol{\theta} \quad \text{Func. approx.} \quad (30)$$

$$[\psi(s_t)]_j = \frac{w_j(s_t)}{\sum_{k=1}^{p} w_k(s_t)} \quad \text{Basis functions} \quad (31)$$

$$w_j = \exp\left(-0.5h_j(s_t - c_j)^2\right) \quad \text{Gaussian kernel} \quad (32)$$

Another perspective on DMPs is that they combine a *feedback* controller f_t with known stability and convergence properties with an *open-loop* forcing term $h_{\boldsymbol{\theta}}$ consisting of a function approximator parameterized with $\boldsymbol{\theta}$.

Equation (26) may be rewritten as in (34), which makes apparent that the first part is a feedback controller (because it takes the current state), whereas the function approximator is an open-loop controller (because it takes only the phase, which is a 1-dimensional representation of time).

$$\frac{1}{\tau}\ddot{x} = \overbrace{\alpha(\beta(x - g) - \dot{x})}^{\text{feedback controller}} + \overbrace{h_{\boldsymbol{\theta}}(s_t)s_t(g - x_0)}^{\text{open-loop controller}} \quad (33)$$

$$= f(x, \dot{x}) + h_{\boldsymbol{\theta}}(s_t)s_t(g - x_0) \quad (34)$$

Here, the open-loop controller is able to represent arbitrary time-dependent movements, whereas the feedback controller makes the resulting motion robust towards perturbations, and ensures convergence towards the goal g.
Multi-dimensional DMPs are acquired by coupling several transformation systems (26) with the same canonical system (29), such that the transformation systems are coupled in time. The output of the D transformation systems may specify for instance the reference trajectories of the D joints of a robot, or the 3D position of a robot end-effector, allowing them to represent for instance a tennis swing, a reaching movement, or a complex dance movement.

5.2. How DMPs Reduce the State Space

In reinforcement learning, the policy maps states to actions (or actions and states to probabilities of executing the action). The "curse of dimensionality" in reinforcement learning refers to the problem that *"the number of parameters to be learned grows exponentially with the size of any compact encoding of a state"* [2].
If we write a D-dimensional DMP as a policy (assuming that g remains constant during the movement for now) $\pi(\mathbf{x}, \dot{\mathbf{x}}, s) \rightarrow \ddot{\mathbf{x}}$, we see that the input state of the policy consists of $\langle \mathbf{x}, \dot{\mathbf{x}}, s \rangle$, which has dimensionality $2 \cdot \dim(\mathbf{x}) + 1$. The output action is $\ddot{\mathbf{x}}$, which has dimensionality $\dim(\mathbf{x})$. For a 7D DMP providing the reference trajectory of the 7 joints of an arm, the state space is therefore 15D, and the action space is 7D. We now discuss two ways in which DMPs reduce the state space $\langle \mathbf{x}, \dot{\mathbf{x}}, s \rangle$.

5.2.1. Transformation Systems are Independent

In DMPs, each of the D transformation systems is integrated independently of the others, and each has its own function approximator h^d and parameter vector $\boldsymbol{\theta}^d$. What synchronizes them in time is the shared phase variable s that arises from integrating the canonical system (29). Thus, the policy $\pi(\mathbf{x}, \dot{\mathbf{x}}, s)$ may be considered to consist of D separate policies $\pi_{\boldsymbol{\theta}^d}(x^d, \dot{x}^d, s)$ which are coupled by s, as in (35).

$$\ddot{\mathbf{x}} \leftarrow \pi(\mathbf{x}, \dot{\mathbf{x}}, s) = \left\{\pi_{\boldsymbol{\theta}^d}(x^d, \dot{x}^d, s)\right\}_{d=1:D} \quad (35)$$

$$\ddot{x}^d \leftarrow \pi_{\boldsymbol{\theta}^d}(x, \dot{x}, s) = \alpha(\beta(x^d - g^d) - \dot{x}^d) + h_{\boldsymbol{\theta}^d}(s_t)s_t(g^d - x_0^d) \quad (36)$$

Analogously, PI^2 optimizes the parameters $\boldsymbol{\theta}^d$ of each of the D function approximators independently of the others. Instead of learning a mapping from $\langle \mathbf{x}, \dot{\mathbf{x}}, s \rangle$ to $\ddot{\mathbf{x}}$, PI^2 with DMPs learns D mappings from $\langle x^d, \dot{x}^d, s \rangle$ to $\ddot{x}^d$. The size of the input space for the former is $2 \cdot \dim(\mathbf{x}) + 1$, and for the latter 3. Thus, D independent optimization processes are run for D different parameter vectors $\boldsymbol{\theta}^d$. What these optimization processes do share is the samples and the costs: each optimization process aims to optimize the same cost function. The advantage of this approach is that this reduces the dimensionality of each optimization process.

5.2.2. Only the Closed-Loop Controller is Learned

A closer look at (34) reveals that different parts of a transformation system use different parts of the 3D input state $\langle x, \dot{x}, s \rangle$: the closed-loop controller $f(x, \dot{x})$ takes $\langle x, \dot{x} \rangle$ as input, and the open-loop controller $h_{\boldsymbol{\theta}}(s)$ takes only the 1-dimensional phase signal s. Because the learned parameters $\boldsymbol{\theta}$ do not influence the closed loop controller, from the learner's point of view, this system may be considered part of the environment (just as the PID controller used to track the reference positions $\mathbf{x}_{\text{ref}}$ is usually not considered part of the policy). This is the same perspective as taken in Equation (25) in [36]. We rewrite (36) as:

$$\ddot{x}^d = \alpha(\beta(x^d - g^d) - \dot{x}^d) + \pi_{\boldsymbol{\theta}^d}(s)s(g^d - x_0^d) \quad (37)$$

$$\pi_{\boldsymbol{\theta}^d}(s) = h_{\boldsymbol{\theta}^d}(s) \quad (38)$$

In (38), the policy thus reduces to the function approximator $h_{\boldsymbol{\theta}}(s)$. Because this is an open-loop controller, it only takes the phase signal as an input. Thus, the state input of the policy has been reduced to 1D, because the phase signal is a representation of time, which is always 1D.

As a consequence, each policy $\pi_{\boldsymbol{\theta}^d}(s) = h_{\boldsymbol{\theta}^d}(s)$ must learn the mapping from a 1D input state s to a 1D output action $\ddot{x}$. This is certainly the most drastic reduction of state space and action space possible! This is because the policy $\pi_{\boldsymbol{\theta}^d}(s)$ is open-loop: it determines an action for each time step, and does not take the state $\langle \mathbf{x}, \dot{\mathbf{x}} \rangle$ into account.

Summary

DMPs tackle the curse of dimensionality in two ways: 1) learn a separate policy for each dimension of the state and action space, independently of the other policies, and 2) reduce the input space and output action space for each of the policies to 1 dimension. Given an D-dimensional state and action space, DMPs thus facilitate learning by reducing a $2 \cdot D + 1$-dimensional state space and D-dimensional action space into D separate learning problems, each with a 1D input/output space.

6. Implications

A clear advantage of DMPs is that the policy for each transformation system maps an *only* 1D input time signal to an *only* 1D output action. Thus, the optimal command becomes time-dependent rather than state dependent. This *dramatically* simplifies learning, because the state may be very high-dimensional, whereas time is always 1-dimensional. As a consequence, the dimensionality of the parameter vector $\boldsymbol{\theta}$ can be expected to be lower than for higher-dimensional input spaces. The reduced dimensionality of the search space leads to quicker convergence.

In fact, this observation begs the question if the success of applying algorithms such as PI^2, PoWER and eNAC to robot skill learning is due to the improvements in these algorithms, or rather the drastic simplification of the learning problem by using DMPs. That a very basic evolutionary algorithm – (μ_W, λ)-ES – is able to outperform these state-of-the-art algorithms seems to indicate the latter, but further theoretical and empirical analysis is required to corroborate this hypothesis.

6.1. Limitations of DMPs, which Enable Simplification of the Learning Problem

Simplifying a learning problem by choosing an appropriate state or policy representation is a valid, often essential, step in making robot skill learning feasible. In the case of DMPs, this consists of learning the parameter of each transformation system independently of the others, and learning the parameters of *only* their open-loop controllers. Although this dramatically simplifies learning, it also places several strong constraints on the types of movements the policy may represent, and the conditions under which it performs optimally. We list some of these constraints below, and discuss recent work that tries to overcome these limitations.

Specific to certain start/end-state

Because g is the attractor of a dynamical system, DMPs can adapt and generalize (on-line) to changing goals. However a guarantee that the DMP will converge to g, does not imply that it will actually achieve the task. The left two graphs in Figure 7 illustrate this. Here, the task is to pass through a viapoint $\mathbf{q}$ with minimal acceleration, the optimal movement for which is shown in the left graph. If the goal g changes to g', the movement will converge to g', but not pass through the viapoint $\mathbf{q}$, thus failing the task. This is because the movement is *only* optimal w.r.t. the initial x_0 and goal g state for which it was optimized. Recent approaches aim at overcoming this limitation by including g and/or x_0 as a task parameter, which we discuss in the next section.

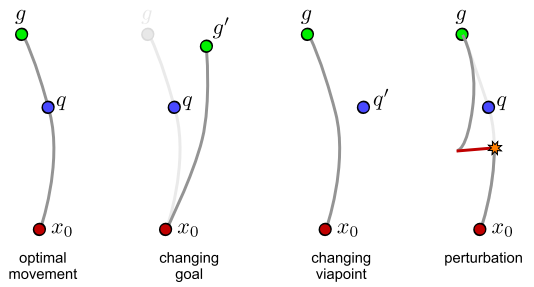

Figure 7. Illustration of lack of optimal adaptation of DMPs to changing goals ($g \rightarrow g'$), task parameters ($\mathbf{q} \rightarrow \mathbf{q}'$) or perturbations.

Specific to certain task parameters

An optimized DMP can usually not adapt optimally to changing tasks. For instance, if the viapoint changes from $\mathbf{q}$ to $\mathbf{q}'$, the DMP will no longer pass through it (third graph in Figure 7). Recent approaches propose, "parameterized skills" [15, 30, 38], which *do* take task parameters into account. They do so by learning a regression between DMP parameters (e.g. $\boldsymbol{\theta}$, g) and task parameters (e.g. the location of the viapoint $\mathbf{q}$). Note that $\mathbf{q}$ usually stays constant during DMP execution. It is therefore not part of the state, but rather considered off-line before movement execution [30]. Thus, the curse of dimensionality applies to the dimensionality of $\mathbf{q}$ (which is usually low, i.e. 1-2 in [15, 30, 38]) and not the dimensionality of the DMP state $\langle \mathbf{x}, \dot{\mathbf{x}}, s \rangle$.

Suboptimal reactions to perturbations

By including the feedback controller in the transformation system, the DMP becomes robust to perturbations, i.e. it will still converge to g. However, as for adaptations to changing g and $\mathbf{q}$, reactions to pertur-

bations cannot be guaranteed to be optimal. The right graph in Figure 7 illustrates this: after a perturbation, the DMP no longer passed through the viapoint $\mathbf{q}$. This is because the open-loop controller, the only component whose parameters are learned, only takes time, but not state, as an input. But state would be necessary to react optimally to perturbations. Therefore recent motion primitive approaches pass state information to the function approximator [14, 17, 18]. Because they include state into the function approximator, they are cursed by the state's dimensionality.

In summary, DMPs have simplified reinforcement learning by dramatically reducing the dimensionality of the state and action space. Solutions to the limitations that have arisen from this simplification rely on expanding the dimensionality of the input space of the function approximator. This leads to the curse of dimensionality, which we believe to be inevitable for any such solution.

7. Conclusion

In this article, we have derived a continuous black-box optimization variant of PI^2, by simplifying the parameter perturbation to be constant (with $\boldsymbol{\epsilon}_k = \boldsymbol{\epsilon}_{k,i}$), temporal averaging to be switched off during the parameter update (with $\delta\boldsymbol{\theta} = \delta\boldsymbol{\theta}_{t_1}$). This variant of PI^2 turns out to be a special case of the evolution strategy (μ_W, λ)-ES (with $\mu = \lambda$ $\sigma = 1$, and $w_k = P_k$). Surprisingly, this simpler variant, which we denote $(\lambda_{W^{PI^2}}, \lambda)$-ES, outperforms PI^2 (and therefore PoWER, eNAC and REINFORCE as demonstrated in [36]) on the five tasks presented by Theodorou et al. [36]. We argue that the main reason for this result is using Dynamic Movement Primitives as the underlying policy representation. This is of particular importance to robot skill learning, where DMPs have become the predominant skill representation. DMPs tackle the curse of dimensionality by learning only an open-loop controller (with a 1D input state) for each dimension of the DMP separately. This drastically simplifies the learning problem, which raises the following question: are recent successes in robot skill learning mainly due to the algorithms or mainly due to the policy representation? The simplification that DMPs provide hinges on adding some constraints on the task being learned, which limit the general applicability of DMPs. We have presented recent related work that aims at overcoming these limitations [14, 15, 17, 18, 30, 38]. In future work, we will conduct further comparisons with gradient-based methods such as eNAC (RL) and NES and PGPE (BBO), as well as evaluations on real-world robotic tasks.

Acknowledgements

We thank Matthieu Geist, Mrinal Kalakrishnan, Jonas Buchli, Nikolaus Hansen and Balázs Kégl for fruitful discussions and suggestions for improvement. We thank Stefan Schaal for providing the source code for running the experiments and tasks in [36]. This work is supported by the French ANR program MACSi (ANR 2010 BLAN 0216 01), http://macsi.isir.upmc.fr

References

[1] L. Arnold, A. Auger, N. Hansen, and Y. Ollivier. Information-geometric optimization algorithms: A unifying picture via invariance principles. Technical report, INRIA Saclay, 2011.

[2] A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event systems*, 13(1-2):41–77, 2003.

[3] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies - a comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[4] L. Busoniu, D. Ernst, B. De Schutter, and R. Babuska. Cross-entropy optimization of control policies with adaptive basis functions. *IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics*, 41(1):196–209, 2011.

[5] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9:937–965, 2008.

[6] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[7] Nikolaus Hansen. The CMA evolution strategy: A tutorial, June 2011. http://www.lri.fr/ hansen/cmatutorial.pdf.

[8] Verena Heidrich-Meisner and Christian Igel. Evolution strategies for direct policy search. In *Proceedings of the 10th international conference on Parallel Problem Solving from Nature: PPSN X*, pages 428–437, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87699-1.

[9] Verena Heidrich-Meisner and Christian Igel. Similarities and differences between policy gradient methods and evolution strategies. In *ESANN 2008, 16th European Symposium on Artificial Neural Networks, Bruges, Belgium, April 23-25, 2008, Proceedings*, pages 149–154, 2008.

[10] A. Ijspeert, J. Nakanishi, P Pastor, H. Hoffmann, and S. Schaal. Dynamical Movement Primitives: Learning attractor models for motor behaviors. *Neural Computation*, 25(2):328–373, 2013.

[11] A. J. Ijspeert, J. Nakanishi, and S. Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.

[12] Shivaram Kalyanakrishnan and Peter Stone. Characterizing reinforcement learning methods through parameterized learning problems. *Machine Learning*, 84(1-2):205–247, 2011.

[13] H.J. Kappen. Path integrals and symmetry breaking for optimal control theory. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(11):P11011, 2005.

[14] S. Mohammad Khansari-Zadeh and Aude Billard. Learning stable non-linear dynamical systems with gaussian mixture models. *IEEE Transactions on Robotics*, 2011.

[15] J. Kober, E. Oztop, and J. Peters. Reinforcement learning to adjust robot movements to new situations. In *Proceedings of Robotics: Science and Systems*, Zaragoza, Spain, June 2010.

[16] J. Kober and J. Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84:171–203, 2011.

[17] D. Marin and O. Sigaud. Towards fast and adaptive optimal control policies for robots: A direct policy search approach. In *Proceedings Robotica*, pages 21–26, Guimaraes, Portugal, 2012.

[18] Mustafa Parlaktuna, Doruk Tunaoglu, Erol Sahin, and Emre Ugur. Closed-loop primitives: A method to generate and recognize reaching actions from demonstration. In *International Conference on Robotics and Automation*, pages 2015–2020, 2012.

[19] J. Peters and S. Schaal. Applying the episodic natural actor-critic architecture to motor primitive learning. In *Proceedings of the 15th European Symposium on Artificial Neural Networks (ESANN 2007)*, pages 1–6, 2007.

[20] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008.

[21] Jan Peters and Stefan Schaal. Reinforcement learning of mo-

tor skills with policy gradients. *Neural networks : the official journal of the International Neural Network Society*, 21(4): 682–97, May 2008. ISSN 0893-6080.
[22] W. B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. Wiley-Blackwell, 2007.
[23] Martin Riedmiller, Jan Peters, and Stefan Schaal. Evaluation of Policy Gradient Methods and Variants on the Cart-Pole Benchmark. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 254–261. IEEE, April 2007. ISBN 1-4244-0706-0. URL
[24] T. Rückstiess, M. Felder, and J. Schmidhuber. State-dependent exploration for policy gradient methods. In *19th European Conference on Machine Learning (ECML)*, 2010.
[25] Thomas Rückstiess, Frank Sehnke, Tom Schaul, Daan Wierstra, Yi Sun, and Jürgen Schmidhuber. Exploring parameter space in reinforcement learning. *Paladyn. Journal of Behavioral Robotics*, 1:14–24, 2010. ISSN 2080-9778.
[26] J.C. Santamaría, R.S. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive behavior*, 6(2):163–217, 1997.
[27] H.-P. Schwefel. *Evolutionsstrategie und numerische Optimierung*. PhD thesis, TU Berlin, 1975.
[28] Frank Sehnke, Christian Osendorfer, Thomas Rückstie , Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.
[29] O. Sigaud and J. Peters. From motor learning to interaction learning in robots. In *From Motor Learning to Interaction Learning in Robots*, volume 264, pages 1–12. Springer-Verlag, 2010.
[30] Bruno Da Silva, George Konidaris, and Andrew Barto. Learning parameterized skills. In John Langford and Joelle Pineau, editors, *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, ICML '12, pages 1679–1686, New York, NY, USA, July 2012. Omnipress. ISBN 978-1-4503-1285-1.
[31] Freek Stulp and Olivier Sigaud. Path integral policy improvement with covariance matrix adaptation. In *Proceedings of the 29th International Conference on Machine Learning (ICML)*, 2012.
[32] Freek Stulp, Evangelos Theodorou, Mrinal Kalakrishnan, Peter Pastor, Ludovic Righetti, and Stefan Schaal. Learning motion primitive goals for robust manipulation. In *International Conference on Intelligent Robots and Systems (IROS)*, 2011.
[33] Freek Stulp, Evangelos Theodorou, and Stefan Schaal. Reinforcement learning with sequences of motion primitives for robust manipulation. *IEEE Transactions on Robotics*, 28(6):1360–1370, 2012. *King-Sun Fu Best Paper Award of the IEEE Transactions on Robotics for the year 2012*.
[34] R. Sutton and A. Barto. *Reinforcement Learning: an Introduction*. MIT Press, 1998.
[35] Minija Tamosiumaite, Bojan Nemec, Ales Ude, and Florentin Wörgötter. Learning to pour with a robot arm combining goal and shape learning for dynamic movement primitives. *Robots and Autonomous Systems*, 59(11):910–922, 2011.
[36] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. A generalized path integral control approach to reinforcement learning. *Journal of Machine Learning Research*, 11:3137–3181, 2010.
[37] Julian Togelius, Tom Schaul, Daan Wierstra, Christian Igel, Faustino Gomez, and Jürgen Schmidhuber. Ontogenetic and phylogenetic reinforcement learning. *Zeitschrift Künstliche Intelligenz - Special Issue on Reinforcement Learning*, pages 30–33, 2009.
[38] Ales Ude, Andrej Gams, Tamim Asfour, and Jun Morimoto. Task-specific generalization of discrete and periodic dynamic movement primitives. *IEEE Transactions on Robotics*, 26(5): 800–815, 2010.
[39] S. Vijayakumar and S. Schaal. Locally weighted projection regression: An o(n) algorithm for incremental real time learning in high dimensional spaces. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*, pages 288–293, 2000.
[40] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, 2008.
[41] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8: 229–256, 1992.

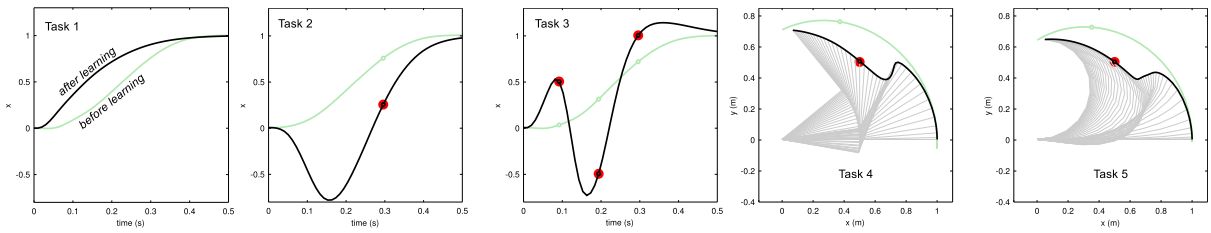

Figure 8. Task 1: Reaching the goal accurately whilst minimizing accelerations before (light green) and after (black) learning. Task 2 and 3: Minimizing the distance to 1 or 3 viapoints before (light green) and after (black) learning. Task 4 (2-DOF) and 5 (10-DOF): minimizing the distance to a viapoint in end-effector space whilst minimizing joint accelerations.

Appendix

A. Evaluation Tasks

We describe the tasks used for the empirical evaluations in Section 4. The implementations are based on the same source code as used for Theodorou et al. [36], and all tasks and algorithms parameters are the same unless stated otherwise. This allows for a direct comparison of the results in this article and those acquired by Theodorou et al. [36]. Due to the similarity, this appendix is very similar to Section 5 of [36], and added for completeness only. PI^2's capability to learn high-dimensional (>10D) problems on physical robot systems has been amply shown in other works [32, 33, 35].

Parameterizations. As in [36], we use Dynamic Movement Primitives as the underlying policy representation in all tasks. The DMPs have 10 basis functions per dimension, and a duration of 0.5s. During learning, $K = 15$ roll-outs are performed for one update[14]. The initial exploration magnitude is $\mathbf{\Sigma} = \lambda \mathbf{I}$ with $\lambda = 0.05$ for all tasks except Task 1, where it is $\lambda = 0.01$. The exploration decay is $\gamma = 0.99$, i.e. the exploration at update u is $\mathbf{\Sigma}_u = \gamma^u \lambda \mathbf{I}$.

Task 1. This task considers a 1-dimensional DMP of duration 0.5s, which starts at $x_0 = 0$ and ends at the goal $g = 1$. In this task as in all others, the initial movement is acquired by training the DMP with a minimum-jerk movement. The aim of Task 1 is to reach the goal g with high accuracy, whilst minimizing acceleration, which is expressed with the following immediate (r_t) and terminal (ϕ_{t_N}) costs:

$$r_t = 0.5 f_t^2 + 5000 \boldsymbol{\theta}^\intercal \boldsymbol{\theta}, \quad \phi_{t_N} = 10000(\dot{x}_{t_N}^2 + 10(g - x_{t_N})^2) \tag{39}$$

where f_t refers to the linear spring-damper system in the DMP, cf. (26). Figure 8 (left) visualizes the movement before and after learning.

Task 2 & 3. In Task 2, the aim is for the output of the 1-dimensional DMP (same parameters as in Task 1) to pass through the viapoint 0.25 at time $t = 300ms$. Which is expressed with the costs:

$$r_{300ms} = 10^8 (0.25 - x_{t_{300ms}})^2, \quad \phi_{t_N} = 0 \tag{40}$$

The costs are thus 0 at each time step except at t_{300ms}. This cost function was chosen by Theodorou et al. [36] to allow for the design of a compatible function for PoWER.

Task 3 is equivalent except that it uses 3 viapoints [0.5 -0.5 1.0] at times [100ms 200ms 300ms] respectively. Figure 8 (2nd and 3rd graphs) visualizes the movement before and after learning for Task 2 and 3.

Note that Task 3 was not evaluated by Theodorou et al. [36]. We have included it as we expected that it is a task where it may be *"helpful to exploit intermediate rewards"* [37], and where RL approaches are conjectured to outperform BBO [37]. As Figure 6 reveals, this is not the case for this particular task, and $(\lambda_{W^{PI^2}}, \lambda)$-ES also outperforms PI^2 for this task.

Task 4 & 5. Theodorou et al. [36] used this task to evaluate the scalability of PI^2 to high-dimensional action spaces and learning problems with high redundancy. Here, an 'arm' with D rotational joints and D links of length $\frac{1}{D}$ is kinematically simulated in 2D Cartesian space. Figure 8 (right two graphs) visualizes the movement by showing the configuration of the arm at each time step. The goal is to pass through a viapoint (0.5,0.5) , in end-effector space, whilst minimizing accelerations. The D joint trajectories are initialized with a minimum-jerk trajectory, and then optimized with respect to the following cost function:

$$r_t = \frac{\sum_{i=1}^{D} (D + 1 - i)(0.1 f_{i,t}^2 + 0.5 \boldsymbol{\theta}_i^\intercal \boldsymbol{\theta}_i)}{\sum_{j=1}^{D} (D + 1 - j)} \tag{41}$$

$$r_{300ms} = 10^8 ((0.5 - x_{t_{300ms}})^2 + (0.5 - y_{t_{300ms}})^2) \tag{42}$$

$$\phi_{t_N} = 0 \tag{43}$$

The weighting term $(D + 1 - i)$ places more weight on proximal joints than distal ones, which is motivated by the fact that proximal joints have lower mass and therefore less inertia, and are therefore more efficient to move [36]. Figure 8 depicts the movements before and after learning for arms with $D = 2$ and $D = 10$ links respectively.

[14] Although 10 roll-outs has usually proven to be sufficient, Theodorou et al. [36] choose 15 roll-outs to allow comparison with eNAC, which requires at least 1 roll-out more than the number of basis functions to perform its matrix inversion without numerical instabilities.