Multi-Level Structure vs. End-to-End-Learning in High-Performance Tactile Robotic Manipulation

Florian Voigt, Lars Johannsmeier and Sami Haddadin
Munich School of Robotics and Machine Intelligence and Chair of Robotics Science and Systems Intelligence
Technical University of Munich Germany
`florian.voigt@tum.de / lars.johannsmeier@tum.de / haddadin@tum.de`

Abstract: In this paper we apply a multi-level structure to robotic manipulation learning. It consists of a hybrid dynamical system we denote skill and a parameter learning layer that leverages the underlying structure to simplify the problem at hand. For the learning layer we introduce a novel algorithm based on the idea of learning to partition the parameter solution space to quickly and efficiently find good and robust solutions to complex manipulation problems. In a benchmark comparison we show a significant performance increase compared with other black-box optimization algorithms such as HiREPS and particle swarm optimization. Furthermore, we validate and compare our approach on a very hard real-world manipulation problem, namely inserting a key into a lock, against state-of-the-art deep reinforcement learning.

Keywords: Reinforcement Learning, Robotic Manipulation, Optimization, Multi-Level Structure

1 Introduction

Torque-controlled robots such as the KUKA LBR iiwa and Franka Emika Panda [1] have significantly impacted the field of robotic manipulation since they can safely interact with their environment based on sensing very small contact forces [2]. However, complex problems can still only be solved by experts who program robots explicitly for specific tasks which is very time consuming. In order to enable robots to work in common industrial or service scenarios on a large scale, machine learning is employed to reduce programming and deployment time significantly [3]. There is much ongoing research in this field e.g. [4, 5, 6]. Many ideas for learning robotic manipulation revolve around dynamic movement primitives (DMP), with some of the most prominent ones being [7, 8, 9]. However, the considered tasks often involve high kinetic energy and aim at achieving periodic tasks such as walking or episodic ones such as batting. The currently most noted approaches are usually based on deep reinforcement learning (RL) [10, 11, 12, 13] and aim at learning manipulation tasks end-to-end, i.e. without assuming a specific structure other than the policy itself. Some authors, e.g. [14] report successes on a variety on manipulation tasks. However, deep RL approaches usually require a very large number of iterations and / or vast computational resources, although very recent work suggests possible future alternatives [15]. Furthermore, much work in this area is done in simulation and has yet to be applied to complex real-world tasks with tight tolerances.

Recently, in [16] a different approach to learning robotic manipulation was proposed based on a structural element denoted as manipulation skill. Such a skill is a multi-level, hybrid dynamical structure which leverages prior existing knowledge about the environment and the robot itself to reduce the complexity of the solution space of the emerging parameter optimization problem. For example, the authors were able to solve challenging manipulation problems such as quickly inserting a cylinder into a hole with industrial tolerances of $<< 0.1$ mm. The learning time took only several minutes while only off-the-shelf computer hardware and black-box-optimization algorithms such as CMA-ES [17] were used.

In this work we extend upon these results by introducing a novel optimization algorithm based on partitioning the solution space employing techniques such as support vector machines (SVM). This enables us to not only find the optimum for a problem but also find robust solutions. The latter property is highly important in robotic manipulation since real-world problems usually have a high degree of noise which may lead to finding solutions that have a low cost but only work in a fraction of the trials.

The contributions of our work are

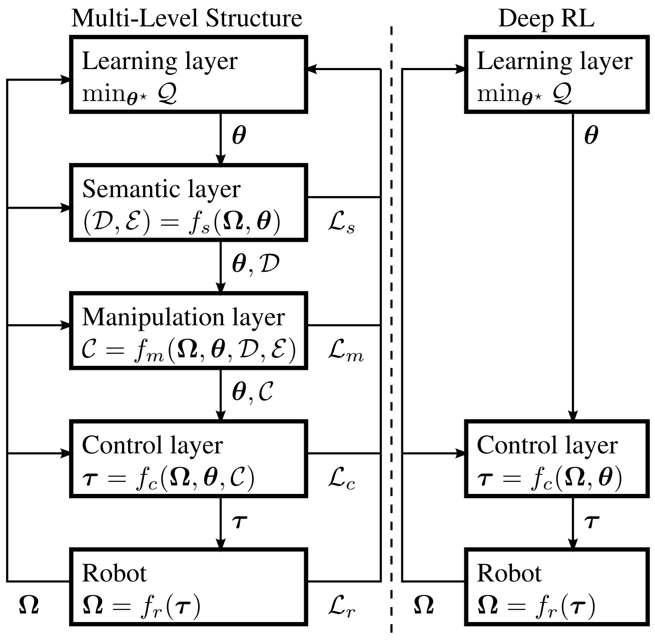
- introduction of a novel parameter optimization algorithm based on learning the solution space partitioning,
- comparison of the algorithm to existing optimization schemes on standard benchmark functions,
- integration of the algorithm into the multi-level structure approach to robotic manipulation learning adapted from [16],
- showcasing the feasibility of the developed algorithm in complex and low-tolerance real-world robotic manipulation problems,
- and comparing the algorithm to state-of-the-art blackbox optimization and deep learning, underlying that our work is able to significantly outperform existing approaches.

The remainder of the paper is organized as follows. In Sec. 2, the overall multi-level approach is explained. Then, in Sec. 3 we present our new algorithm in detail. We also discuss additional algorithms and evaluate and compare them on two benchmarking problems in Sec. 4. Subsequently, in Sec. 5 the real-world experiment is presented and the results are discussed. Finally, Sec. 6 concludes the paper.

2 Multi-Level Structure Learning and Deep RL

In this section we show the multi-level structure based on [16] in more detail and put it side-by-side with the structure of a deep RL approach to highlight the principle differences. This structural comparison supplements the experimental comparison in Sec. 5.

The multi-level structure is descending from a high-level learning layer down to a nonlinear controller that is directly connected to the physical robot platform, see Fig. 1 on the left side.



Multi-Level Structure
Deep RL
Learning layer $\min_{\boldsymbol{\theta}^\star} \mathcal{Q}$
$\boldsymbol{\theta}$
Semantic layer $(\mathcal{D},\mathcal{E}) = f_s(\boldsymbol{\Omega},\boldsymbol{\theta})$
$\mathcal{L}_s$
$\boldsymbol{\theta},\mathcal{D}$
Manipulation layer $\mathcal{C} = f_m(\boldsymbol{\Omega},\boldsymbol{\theta},\mathcal{D},\mathcal{E})$
$\mathcal{L}_m$
$\boldsymbol{\theta},\mathcal{C}$
Control layer $\boldsymbol{\tau} = f_c(\boldsymbol{\Omega},\boldsymbol{\theta},\mathcal{C})$
$\mathcal{L}_c$
$\boldsymbol{\tau}$
Robot $\boldsymbol{\Omega} = f_r(\boldsymbol{\tau})$
$\mathcal{L}_r$
$\boldsymbol{\Omega}$
Learning layer $\min_{\boldsymbol{\theta}^\star} \mathcal{Q}$
$\boldsymbol{\theta}$
Control layer $\boldsymbol{\tau} = f_c(\boldsymbol{\Omega},\boldsymbol{\theta})$
$\boldsymbol{\tau}$
Robot $\boldsymbol{\Omega} = f_r(\boldsymbol{\tau})$
$\boldsymbol{\Omega}$

Figure 1: The multi-level structure approach.

The learning layer employs a schema that minimizes a cost function by finding an optimal set of parameters i.e. $\min_{\boldsymbol{\theta}^\star} \mathcal{Q}$. This is done by selecting different sets of parameters $\boldsymbol{\theta}$ trial-by-trial until the optimal parameters $\boldsymbol{\theta}^\star$ are found. During this process each trial's parameter sets are passed to the semantic layer that decides whether the skill can be executed in the first place, when it is successful or whether it fails. Furthermore, it maintains internal models about the environment $\mathcal{E}$. It can be expressed by $(\mathcal{D},\mathcal{E}) = f_s(\boldsymbol{\Omega},\boldsymbol{\theta})$. $\mathcal{D}$ represents the decisions made by the semantic layer and $\boldsymbol{\Omega}$ the percept vector which is fed back from the lower layers, containing the robot state and external environment information. The manipulation layer yields continuous commands $\mathcal{C}$ that are based on the percept vector, the parameters and the decisions made by the semantic layer i.e. $\mathcal{C} = f_m(\boldsymbol{\Omega},\boldsymbol{\theta},\mathcal{D},\mathcal{E})$. On the lowest level, the controller receives the commands from the manipulation layer and outputs torques to the robot i.e. $\boldsymbol{\tau} = f_c(\boldsymbol{\Omega},\boldsymbol{\theta},\mathcal{C})$. Typical controllers are impedance [18] and force controllers [19] or adaptive versions of them [20]. Every layer passes information on the constraints it imposes on the solution space up to the learning layer. These constraints are composed of dynamic robot system limits $\mathcal{L}_r$, control law restrictions $\mathcal{L}_c$ from the control layer, limitations on parameters due to specific strategies $\mathcal{L}_m$ and constraints $\mathcal{L}_s$ coming from reasoning about the environment and internal models.

Figure 2 depicts the overall complexity reduction behind the considered multi-level approach. The very complex solution space of all possible motor commands is step-wise reduced by the well designed different computational layers to a small and tractable subset that appears to be a promising candidate set. The right side of Fig. 1 depicts the structure of a deep RL approach. The notable difference to the multi-level approach is that deep RL algorithms make little to no assumptions about the inner workings of their policy.

3 Parameter Space Partitioning

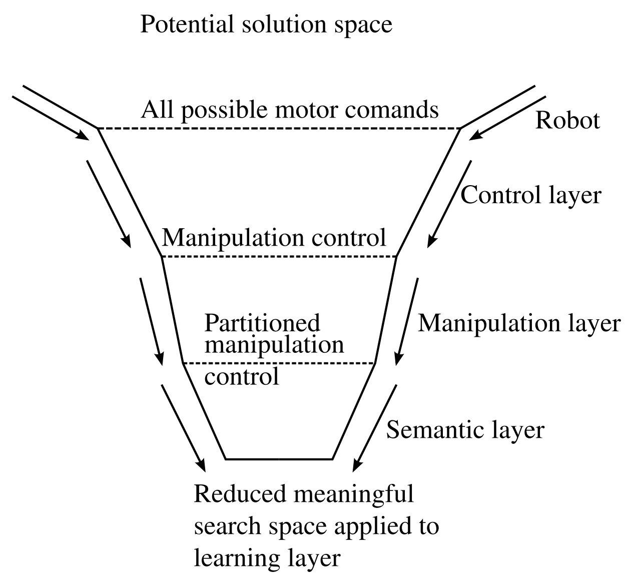

Potential solution space
All possible motor comands
Robot
Control layer
Manipulation control
Manipulation layer
Partitioned manipulation control
Semantic layer
Reduced meaningful search space applied to learning layer

Figure 2: Visual representation of the solution space reduction by application of the multi-level approach.

In this section we introduce a novel algorithm, the parameter space partitioning (PSP) algorithm. First, we will outline its structure and then compare it on its own to other optimization algorithms in the next section. In Sec. 5 we integrate it into the learning layer of the multi-level approach described in Sec. 2.

The goal of the algorithm is to find a set of optimal parameters $\theta^\star$ that minimizes a given cost function $\mathcal{Q}$.

Following Sec. 2 we start with a reduced parameter space, see Fig. 2, of dimension n. Since the constraints of this space are known we can rephrase the problem as finding optimized tuples in a hypercube. We do not make any assumptions on minima and cost function convexity, assume that we have no gradients and noisy, inherently stochastic problems. We assume that the solution space is at least piece-wise continuous in terms of the cost function. Subsequently, we can assume that the according hypercube can be separated by a reward r_{sep} into spaces which yield rewards $r_i \geq r_{sep}$ and spaces which yield $r_i < r_{sep}$. This can be done by e.g. a Support Vector Machine. If we increase r_{sep} iteratively, we obtain an approach to find optima within this solution space. The PSP algorithm consist of three elements: A proposal policy $p(a)$, a filtering policy $f(p_p)$ and an update step. It's core principle is to partition the hypercube of the solution space, which is shown in the following.

3.1 General Algorithm Outline

The PSP algorithm consists of k episodes. Each episode consists of a number of trials n_ε. The learning process is as follows: For each trial i of an episode, parameters θ_{s_i} are sampled $\propto q(a)$ in sample-space, i.e. the hypercube, with $q(a)$ being the sampling policy. These are translated into solution-space and applied to the optimization problem. The resulting reward r_i is stored together with the parameters in sample-space θ_{s_i}. If a trial is unsuccessful, we set the reward r_i to a negative value, $r_i = -1$. This is done to assure negative classification in the update step. At the end of each episode, the sampling policy $q(a)$ is updated. The sampling policy $q(a)$ consists of two elements: A proposal policy $p(a)$ and a filtering policy $f(p(a))$. $p(a)$ proposes parameters until a sample has been found which is accepted by the filtering policy $f(p(a))$. In the following subsections, $q(a)$, $p(a)$, $f(p(a))$ and the update step are presented in detail.

3.2 Sampling Policy $q(a)$

The key element of our algorithm, the sample policy $q(a)$, is explained in the following. It consists of two parts: A proposal policy $p(a)$ and a filtering policy $f(p(a))$. $p(a)$ proposes parameters θ_p, which are then classified by the filtering policy $f(\theta_p)$. The filtering policy is a non-linear Support Vector Machine. Both will be discussed in detail in 3.2.1 and 3.2.2. Due to the fact that it is impossible to generate samples directly from SVMs, we use the proposal policy $p(a)$ to generate samples, which are then classified by the filtering policy. In the beginning, $p(a) = lhs(0,1)^n$ is a latin hypercube sampler of dimension n and the SVM is $not\ active$, due to lack of data. If the SVM is $not\ active$, especially in the first episode, the parameters $\theta_p \propto p(a)$ are applied directly to the experiment. If the SVM is $active$, proposed parameters θ_p are classified, and if found to be positive/potentially useful, applied to the optimization problem. θ_p is a proposed parameter-tuple $\propto p(a)$ which is evaluated by the SVM. n_s is the maximum number of proposals per trial. If n_s is exceeded, i.e. no sample which satisfies $f(p(a))$ has been found in reasonable time, $\theta_p \propto p(a)$ is applied to the problem regardless.

Algorithm 1 Parameter Space Partitioning Algorithm

```
**Input:** Number of episodes $k$, Length of episode $n_\varepsilon$, Minimum number of successful trials for
proposal $n_{p,\min}$, for filtering $n_{f,\min}$
**Initialize:** $\bar{r} = 0,\ q(a) = lhs(0,1)^p$
**for** $1:k$ **do**
    **Samples:** generate $n_\varepsilon$ samples from sample policy $q(a)$ and add to data
    $\{\theta_{s_i} \sim q(a), r_i\} i \in \{1, ..., n_\varepsilon\}$
    **Update Sample Policy** $q(a)$**:**
    $\forall r_i < \bar{r} \rightarrow s_i = 0, \forall r_i \geq \bar{r} \rightarrow s_i = 1, \bar{r} = \frac{1}{\sum s_i} \sum r_i s_i$
    Update proposal policy $p(a)$:
    iff $|\forall r_i > 0| \geq n_{p,\min} \rightarrow p(a) \propto GMM(a)$
    else $p(a) \propto \mathcal{U}(a)$
    Update filtering policy $f(p)$:
    iff $|\forall r_i > 0| \geq n_{f,\min} \rightarrow f(p) \propto SVM(p)$
    else $\nexists f(p)$
```

3.2.1 Proposal Policy $p(a)$

For our proposal policy we evaluate three different sampling methods: Latin hypercube sampling, random uniform sampling and Gaussian Mixture Models. These are introduced in the following:

- Latin Hypercube Sampling (LHS): LHS [21] is a statistical method for generating samples. Each dimension of the sampling space is divided into n_ε equidistant parts. Then the n_ε sampling-points are distributed in a manner, that only one sample per division and dimension occurs.
- Random Uniform Sampling: Parameters are sampled with equal probability over an n-dimensional space.
- Gaussian Mixture Model(GMM): GMMs [22] consist of multiple (multivariate) Gaussians, which form a distribution. Each Gaussian has a weight k assigned to it which specifies the probability of being sampled from.

In the beginning, the proposal policy $p(a)$ is initialized with an LHS $p(a) = lhs(0,1)^n$, with n being the number of parameters. The samples are generated $\propto n_\varepsilon$, the number of tuples per episode. This is to promote a more evenly distribution in sample space. After the first episode, $p(a)$ is updated to a uniform random sampler $\sim \mathcal{U}(0,1)^n$. In later episodes, when its unlikely that the proposal policy is able to come up with useful samples in reasonable time, a standard Gaussian mixture model (GMM) is used, sufficient data presumed. This is achieved, when the number of successful samples is $\geq n_{p,\min}$. While this does have slight influence on the learning behavior, our intention is solely to limit time consumption. In order to obtain a useful GMM, we train several GMMs with different numbers of components in the range of $1 - n_g$. n_g is a specified maximum number of options, i.e. different solutions. For each GMM, the Bayesian Information Criterion (BIC) is calculated and the GMM with the lowest BIC is chosen as $p(a)$. Note that we only take successful parameter samples into consideration when calculating the GMMs (successful in the sense, that the task goal is achieved). The number of components stay constant for the rest of the learning process, in an attempt to counter premature convergence of different options to a single one. Due to sparse data, GMMs with many components might be distributed badly. As we deal with reinforcement learning, our data is neither homogeneously distributed nor comprehensive. This makes it difficult to obtain well defined GMMs. Often, when only a single optimized solution is wanted, n_g should be in the range of $1 - 3$, depending on the complexity of the optimization problem.

3.2.2 Filtering Policy $f(p)$

As the filtering policy $f(p)$, we use a non-linear SVM with rbf-kernels and high cost C, i.e. we penalize misclassification strictly. Also, we demand a minimum number of successful samples $n_{f,\text{min}}$ for a robust estimation. As long as this criterion is not met, we do not use the filtering policy. Useful values for $n_{f,\text{min}}$ might be problem-dependent. It is our goal to obtain a decision function which separates successful and unsuccessful samples equidistantly. In order to obtain a useful version of the SVM, we train for several different γ, i.e. kernel sizes. Then we calculate the minimum distance over all samples to decision plane for each SVM. Subsequently, we choose the SVM which yields the highest minimum distance, i.e. the best fit for our purpose, as $f(p)$. This allows us to divide even complex distributions accordingly.

3.3 Updating sampling policy $q(a)$

Lastly, we need to discuss our approach to updating the obtained data. After each episode, the mean reward $\bar{r}$ is updated and the available samples are marked as successful/unsuccessful. The samples θ_{s_i} have an according reward r_i. All samples $< \bar{r}$ are set to unsuccessful and the mean $\bar{r}$ is recalculated for all remaining successful samples. Note that we draw the line specifically at $< \bar{r}$. This is done for the case that only one successful sample remains, which would also be labelled unsuccessful due to not yielding a higher reward than the mean. Lastly, the sampling policy $q(a)$, i.e. $p(a)$ and $f(p)$, is recalculated based on the updated data. The initial mean reward $\bar{r}$, which is used to classify samples, is set to $\bar{r} = 0$. While this approach is similar to the Cross-Entropy Method [23], [24], we do not have a fixed amount or percentage of champions we use to calculate the distribution and are not neglecting the information of the other samples due to the discriminative nature of the partitioner.

3.4 Visualization of algorithm

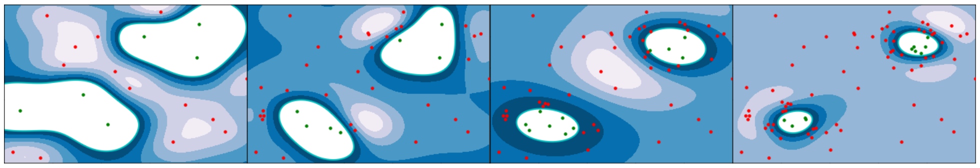

(a) 1st episode (b) 2nd episode (c) 3rd episode (d) 4th episode

Figure 3: Learning behavior for a 2D problem

Fig. 3 shows the exemplary learning behavior of the algorithm over multiple episodes. The optimization problem is as follows:

$$r = e^{-((x-x_1)^2+(y-y_1)^2)} + e^{-((x-x_2)^2+(y-y_2)^2)} \tag{1}$$

with

$$x_1 = y_1 = 0.25 \text{ and } x_2 = y_2 = 0.75 \tag{2}$$

and r is the reward to be maximized. We use 20 trials per episode and observe 4 episodes in total. The red dots are parameter-samples labeled as unsuccessful and the green ones are defined as successful. The white spaces are positive i.e. desirable sub-spaces of parameters and the blueish ones are unfavourable. In the first episode (3a), latin-hypercube-sampling is applied which results in a good distribution of samples over search-space. Based on this, we obtain a first distinction between useful and useless spaces for continuing our search via the SVM. In the second episode (3b) we can see a convergence of our desired search-space. It is noticeable that new parameter-samples were added within the previous search-spaces. In the 3rd (3c) and 4th (3d) episode, this convergence continues and the two optima of the optimization problem have been narrowed down. One can see that our learner is a multimodal optimizer, such as [25] and [26]. In contrast to those, we incorporate a discriminative nature via the SVM, which explicitly punishes unsuccessful/subpar samples.

4 Simulation Benchmarks

In this section, we first briefly introduce state of the art blackbox algorithms we compare against on a typical simulation benchmark. Then, we present two benchmark problems we used for reference and discuss the results. The following blackbox algorithms were implemented and applied: HiREPS [27], Genetic Algorithm (GA) [28] and Particle Swarm Optimization (PSO) [29]. A detailed overview of the algorithms and their implementation is available in appendix A.

4.1 Benchmark Problems

To benchmark the introduced algorithms, we take the Sphere-function and the Rastrigin-function. For each optimization problem, multiple dimensions ranging from 5 to 20 parameters are considered. Goal is the minimization of cost C, which is defined in the following. n is the number of parameters and θ_{t_i} the i-th parameter in task-space. The episode length $n_\varepsilon = 40$, the number of episodes $k = 10$ and the number of experiments $n_{exp} = 50$ for both algorithms.

The Sphere function is a convex problem with one global minimum and no local ones:

$$C_i = \sum_{n}^{j=1} \theta_{t_{i,j}}^2 \tag{3}$$

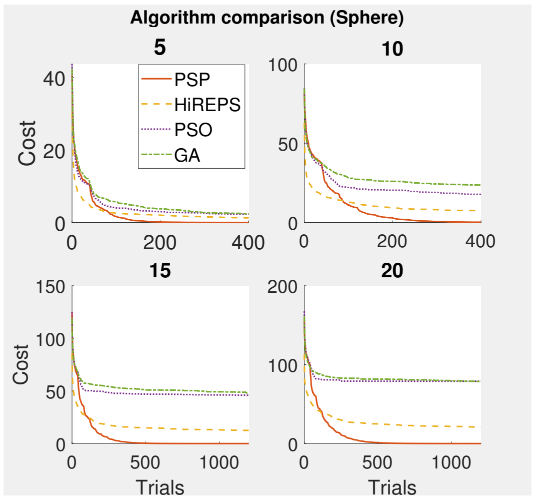
The Rastrigin function is a concave problem with one global minimum and numerous local ones.

$$C_i = An + \sum_{n}^{j=1} [\theta_{t_{i,j}}^2 - Acos(2\pi\theta_{t_{i,j}})] \tag{4}$$

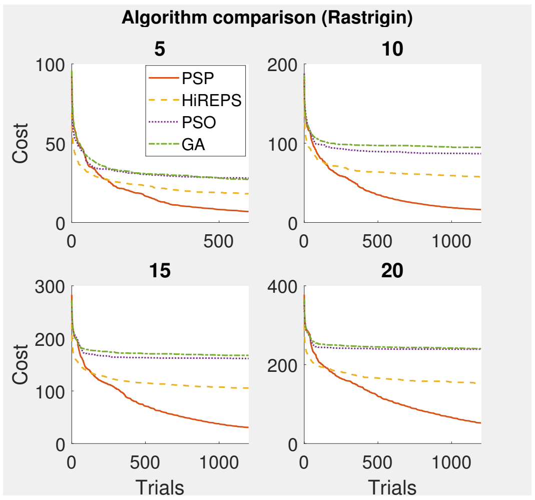
Here $A = 10$ and for both functions the search space is bounded by:

$$-5 \leq \theta_{t_{i,j}} \leq 5 \tag{5}$$

Fig. 4a and fig. 4b show a comparison for the different numbers of parameters. Our algorithm fares significantly better than the others and shows no signs of saturation. HiREPS shows better performance in the early stages, but this is rather a result of undesired bias due to the way we initialize the sampling distribution, instead of reliable superiority. Also one can see the similarity of PSP, PSO and GA for the first episode, due to all being initialized via random uniform distribution.



(a) Sphere problem

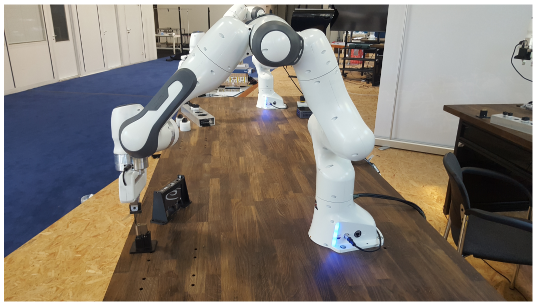


(b) Rastrigin problem

Figure 4: Comparison for the Sphere and Rastrigin problem

5 Experiments

Finally, we tested our algorithm within a real-world application. Based on the insertion skill defined in [16], we aim to learn key insertion, see Fig. 5. The task is carried out using a Franka Emika Panda arm (Fig. 5a). Figure 5b shows a close up of the used key and lock. The following algorithms are applied for benchmarking: HiREPS, CMAES, our PSP algorithm and four deep RL-based algorithms, namely DDPG [30], A3C [31], TD3 [32] and SAC [33]. While we are aware that hybrid approaches between RL and conventional feedback control, such as Residual RL [34] exist, we focus rather on the comparison between parametrized skills and pure RL. The episode length is set to 20. Note that we do not investigate the genetic algorithm and particle swarm optimization as we may refer to [16]


(a) Franka Emika Panda

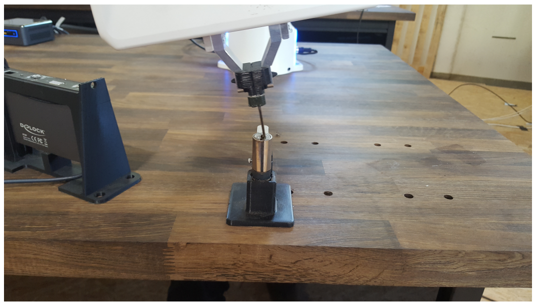

(b) Key-in-Lock task

Figure 5: Key-in-Lock experimental setup for validating the multi-level structure learning against state of the art algorithms

for the details of this cost. The goal is to optimize the task in terms of execution time leading to the cost function

$$C_i = \int 1dt + g\Delta z. \tag{6}$$

In (6) Δz is the distance of the robots end-effector from the target insertion depth in z-direction and g is a multiplication factor. The time taken for cost calculation is the time of the insertion process only, neglecting approach phase and extraction. The neural network based learners are also applied to the insertion phase only in order to ensure comparability. This temporal abstraction approach is similar to [35], with the difference that we assume the approach/search phase to be solved already. While the same cost function is applied for comparison, the learners use the following reward function:

$$r(s, a) = g\Delta z_a \tag{7}$$

with Δz_a being the change of distance to the lock in z-direction between previous state s_i and resulting state s_{i+1}. We get the state and apply a new action each 100 ms. Our PSP algorithm and HiREPS are applied to the multi-level skill-based approach as described in Sec 2. In this experiment, the skill features 23 parameters, which need to be optimized, such as insertion angle, frequency of wiggling and applied feed-forward force. The neural network based algorithms do not use the skill definition but learn torques in joint-space in order to solve the task. The used action space is of dimension 7 as we learn the torques per joint. The observation space is of dimension 21, namely q,$\dot{q}$ and τ_{ext}. For each algorithm, we carried out 10 experiments, consisting of 10 episodes, which had 20 trials each. So in total, for each learner 2000 trials were carried out. The success condition is defined as reaching a defined insertion depth while staying within a region of interest (ROI) of radius 5 mm around the lock. Failure occurs if either the ROI is left, τ_{ext} violates predefined limits or if the insertion process exceeds 5 s of execution time.

5.1 Results

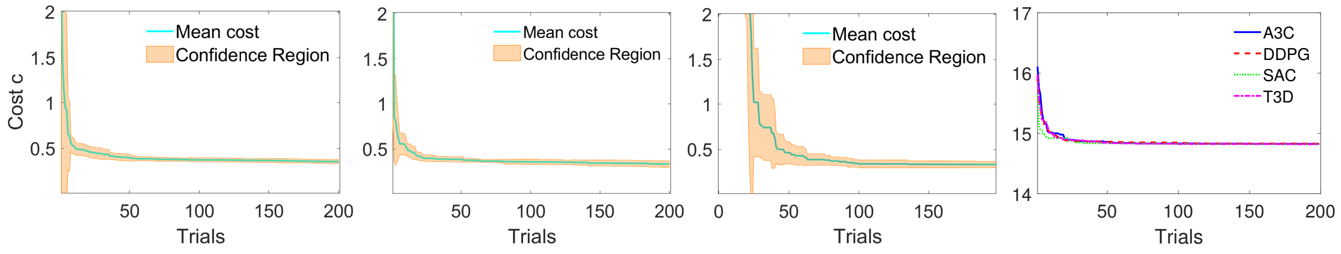
Figure 6 and Tab. 1 show the results for the applied algorithms. None of the neural network based learners was able to solve the task in our experiments. Figures 6a and 6b show clear learning behavior for PSP and HiREPS which seem to be similar. CMAES, while also reaching a similar optimized cost, shows a distinctively slower learning rate (fig. 6c). Table 1 shows the average optimized cost over 10 experiments.

	PSP	HiREPS	CMAES	DDPG	TD3	SAC	A3C
result	success	success	success	failure	failure	failure	failure
$\bar{c}_o$	0.3514	0.3345	0.3384	14.842	14.816	14.834	14.831
$\mathsf{var}(\bar{c}_o)$	0.0225	0.0366	0.0377	0.136	0.193	0.163	0.118

Table 1: Optimized costs of Key-in-Lock experiment for successful trials

5.2 Interpretation

Both PSP and HiREPS seem to fare similar in the real-world experiment. Interestingly, the variance of HiREPS is higher, while the average cost is slightly lower. This stems from the fact that HiREPS finds unreliable solutions, which can yield slightly higher rewards but are prone to failure or yielding



(a) PSP (b) HiREPS (c) CMAES (d) Deep RL

Figure 6: Costs of Key-in-Lock experiment

far lower rewards. In order to verify this, we calculated the mean and variance of the parameters for those rewards and checked the data for parameter tuples within this range. We could verify that for those parameter tuples the probability of yielding those superior costs was around 38 %, while 62 % resulted in far higher costs or failure. The same investigation applied to PSP did not result in such findings. Tab. 2 shows the average percentage of successful trials per episode over the course of learning. One can see that our algorithm is able to discriminate between successful and unsuccessful parameter spaces and therefore finds reliable optimized solutions. Also, due to its nature, it is able to not only provide a singular solution, but rather a solution subspace within our initial search space. CMEAS needs several episodes to achieve a distinctive percentage of successful proposals, but still only manages to get on par with HiREPS.

Episode	1	2	3	4	5	6	7	8	9	10
PSP	50.7	71.2	85.5	89.8	91.2	94.2	96.2	98.5	98.3	98.8
HiREPS	57.5	54.5	60.5	61.5	69.5	67.5	72.0	71.5	69.0	69.5
CMAES	2.8	21.1	43.3	59.4	71.1	65.0	71.1	68.9	71.7	69.2

Table 2: Successful trials over episodes in % (averaged over experiments)

5.3 Discussion

Both benchmark problems and experiments show that our algorithm is suitable to find and optimize parameters for black-box problems and real-world robotic manipulation tasks. Our algorithm fares significantly better in simulations and benchmark problems, i.e. deterministic environments, than the other black-box-optimization algorithms. In the real world, i.e. problems which are not deterministic in general or feature a significant amount of noise or uncertainty, we are able to find reliable and robust solutions quickly, see Sec. 5.2 and Tab. 2. Not only are our converged solutions robust, but also the ones sampled directly from the learner, which allows to improve while achieving success to a certain margin. The general idea of defining and iteratively reducing subspaces of interest seems a promising direction and allows us to not only optimize parameters, but also approximate subspaces of parameters which are applicable to a specific optimization problem or yield a specified reward. We also compared our approach to the state of the art in robotic manipulation learning which is not able to find solutions to the considered (very challenging) manipulation problem, at least within the given frame of 200 trials.

These findings suggest that employing a meaningful structure may be beneficial for learning very challenging manipulation problems such as key insertion.

6 Conclusion

Finding and optimizing parameters for robotic manipulation tasks is tedious and often too complex to be done by hand. In this paper, we extended the structure-based manipulation learning approach by a Support Vector Machine based algorithm on the parameter learning level. We compared this system to other well known learning approaches based on simulation benchmark problems where our solution showed significantly higher performance. Furthermore, we compared the overall system with state-of-the-art manipulation learning approaches in a complex real-world task i.e. inserting a key into a lock. Our approach outperformed the state of the art deep-RL-based approaches and showed interesting properties in terms of robustness when compared to algorithms such as HiREPS. In future work we intend to further develop the algorithm by using a partitioner based on a Neural Network instead of an SVM.

Acknowledgments

Funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy – EXC 2050/1 – Project ID 390696704 – Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden. We gratefully acknowledge the funding of project KoBo34 (Verbundprojektnummer V5ARA202) by the BMBF (grant no. 16SV7985) and the companies Microsoft and Vodafone for this research. We also thank the reviewers for their valuable input and ideas for future work.

References

[1] S. Haddadin, S. Haddadin, and S. Parusel. Franka emika gmbh, 2017. URL www.franka.de.

[2] S. Haddadin, A. De Luca, and A. Albu-Schäffer. Robot collisions: A survey on detection, isolation, and identification. *IEEE Transactions on Robotics*, 33(6):1292–1312, 2017.

[3] J. Peters. Machine learning for motor skills in robotics. *KI-Künstliche Intelligenz*, 2008(4): 41–43, 2008.

[4] S. Calinon, P. Kormushev, and D. G. Caldwell. Compliant skills acquisition and multi-optima policy search with em-based reinforcement learning. *Robotics and Autonomous Systems*, 61 (4):369–379, 2013.

[5] C. Daniel, G. Neumann, and J. Peters. Learning concurrent motor skills in versatile solution spaces. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3591–3597. IEEE, 2012.

[6] M. P. Deisenroth and C. E. Rasmussen. Efficient reinforcement learning for motor control. In *10th International PhD Workshop on Systems and Control*. Hluboka nad Vltavou, 2009.

[7] S. Schaal, J. Peters, J. Nakanishi, and A. Ijspeert. Learning movement primitives. In *Robotics Research. The Eleventh International Symposium*, pages 561–572. Springer, 2005.

[8] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[9] P. Geibel. Reinforcement learning for mdps with constraints. In *European Conference on Machine Learning*, pages 646–653. Springer, 2006.

[10] S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

[11] T. Inoue, G. De Magistris, A. Munawar, T. Yokoya, and R. Tachibana. Deep reinforcement learning for high precision assembly tasks. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 819–825. IEEE, 2017.

[12] A. A. Rusu, M. Večerík, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell. Sim-to-real robot learning from pixels with progressive nets. In *Conference on Robot Learning*, pages 262–270, 2017.

[13] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, et al. Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on Robot Learning*, pages 651–673, 2018.

[14] S. Levine, N. Wagener, and P. Abbeel. Learning contact-rich manipulation skills with guided policy search. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 156–163. IEEE, 2015.

[15] A. Petrenko, Z. Huang, T. Kumar, G. Sukhatme, and V. Koltun. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. *arXiv preprint arXiv:2006.11751*, 2020.

[16] L. Johannsmeier, M. Gerchow, and S. Haddadin. A framework for robot manipulation: Skill formalism, meta learning and adaptive control. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 5844–5850. IEEE, 2019.

[17] N. Hansen. The cma evolution strategy: a comparing review. *Towards a new evolutionary computation*, pages 75–102, 2006.

[18] N. Hogan. Impedance control: An approach to manipulation. In *American Control Conference, 1984*, pages 304–313. IEEE, 1984.

[19] B. Siciliano and L. Villani. *Robot force control*, volume 540. Springer Science & Business Media, 2012.

[20] G. Ganesh, A. Albu-Schäffer, M. Haruno, M. Kawato, and E. Burdet. Biomimetic motor behavior for simultaneous adaptation of force, impedance and trajectory in interaction tasks. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2705–2711. IEEE, 2010.

[21] M. D. McKay. Latin hypercube sampling as a tool in uncertainty analysis of computer models. In *Proceedings of the 24th conference on Winter simulation*, pages 557–564, 1992.

[22] D. A. Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741, 2009.

[23] P. Boer, D. Kroese, S. Mannor, and R. Rubinstein. A tutorial on the cross-entropy method. *Ann Oper Res*, 134, 07 2002.

[24] D. Kroese, S. Porotsky, and R. Rubinstein. The cross-entropy method for continuous multi-extremal optimization. *Methodology and Computing in Applied Probability*, 8:383–407, 09 2006. doi:10.1007/s11009-006-9753-0.

[25] T. Osa. Multimodal trajectory optimization for motion planning. *The International Journal of Robotics Research*, 39:027836492091829, 06 2020. doi:10.1177/0278364920918296.

[26] F. Streichert, G. Stein, H. Ulmer, and A. Zell. A clustering based niching ea for multimodal search spaces. volume 2936, pages 293–304, 10 2003. doi:10.1007/978-3-540-24621-3_24.

[27] C. Daniel, G. Neumann, and J. Peters. Hierarchical relative entropy policy search. In *Artificial Intelligence and Statistics*, pages 273–281, 2012.

[28] L. Davis. Handbook of genetic algorithms. 1991.

[29] R. Eberhart and J. Kennedy. Particle swarm optimization. In *Proceedings of the IEEE international conference on neural networks*, volume 4, pages 1942–1948. Citeseer, 1995.

[30] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[31] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[32] S. Fujimoto, H. Van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[33] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[34] T. Johannink, S. Bahl, A. Nair, J. Luo, A. Kumar, M. Loskyll, J. A. Ojea, E. Solowjow, and S. Levine. Residual reinforcement learning for robot control. *CoRR*, abs/1812.03201, 2018. URL `http://arxiv.org/abs/1812.03201`.

[35] T. Inoue, G. D. Magistris, A. Munawar, T. Yokoya, and R. Tachibana. Deep reinforcement learning for high precision assembly tasks. *CoRR*, abs/1708.04033, 2017. URL `http://arxiv.org/abs/1708.04033`.

A Appendix

A.1 HiREPS

The Hierarchical Relative Entropy Policy Search algorithm was proposed in [27]. Here, the design is given only schematically. The algorithm was implemented and applied to the optimization problem at hand.

Algorithm 2 Hierarchical Relative Entropy Policy Search

```
Input: Information loss tolerance $\varepsilon$, Entropy tolerance $\kappa$, Number of options $n$
Initialize: $\pi$ using $n$ Gaussians with random mean
for $k = 1 : L$ do
    Set sample policy:
    $q(a|s) = \sum_{\mathcal{O}} \pi_{old}(\mathcal{O}|s)\pi_{old}(a|s,\mathcal{O})$
    Samples: collect new samples from the sample
    policy and add to dataset
    $\{s_j \sim p(s_0), a_j \sim q(a|s_j), R_j\} j \in \{1, ..., N\}$
    Calculate importance weights:
    $v_i^k = \frac{q_k(s_i,a_i)}{\sum_{h=k-H}^{k} q_h(s_i,a_i)}$ for all $i$
    Proposal distribution:
    $\widetilde{p}(\mathcal{O}|s_i,a_i) = p_{old}(\mathcal{O}|s_i,a_i)$ for all $i$
    Minimize dual function:
    $[\theta,\eta,\epsilon] = \arg\min_{[\theta,\eta,\epsilon]} g(\theta,\eta,\epsilon)$
    Policy Update:
    $p(s_i,a_i,\mathcal{O}) \propto v_i^k \, \widetilde{p}(\mathcal{O}|s_i,a_i)^{1+\epsilon/\eta} \exp\left(\frac{R_i-\theta^T\phi(s_i)}{\eta}\right)$
    estimate $\pi(\mathcal{O}|s)$ and $\pi(a|s,\mathcal{O})$
    Output:Policy $\pi(a,\mathcal{O}|s)$
```

A.2 Genetic Algorithm

Genetic Algorithms(GA) [28] are inspired by natural selection and behavior of genetics. GA is a branch of Evolutionary Computation. A population of parameter-tuples is updated iteratively via recombination, mutation and evaluation of via a fitness-function. The following implementation was found practically:

$$g_{i,x_{new}} = g_{i,x_{old}} + \mathcal{N}(0; \sigma_x^2 = 0.05) \tag{8}$$

$$p = \begin{cases} p_1 & \text{if} \quad v <= 0.25 \\ p_1 + 2(v - 0.25)(p_2 - p_1) & \text{if} \quad 0.25 < v < 0.75 \\ p_2 & \text{if} \quad v >= 0.75 \end{cases} \tag{9}$$

$g_{i,x}$ is the x-th parameter of the i-th tuple, with x being chosen randomly. p_1,p_2 are the two parameter-tuples chosen for crossover.

A.3 Particle Swarm Optimization

Particle Swarm Optimization(PSO) is an iterative algorithm inspired by the behavior of birds. Initially, several particles are spawned randomly in search-space. Then, iteratively, their velocity is updated over the experiment. Extensive coverage may be found in [29] and [**?**]. During our experiments, we found, that a global velocity adaption worked best with the following velocity adaption law:

$$\mathcal{V}_i = \mathcal{W}_i \mathcal{V}_i + c_1 r_1 (p_i - x_i) + c_2 r_2 (p_g - x_i) \tag{10}$$

$$x_i = x_i + \mathcal{V}_i. \tag{11}$$

Here, $\mathcal{V}_i$ is the velocity of particle i, $\mathcal{W}_i = 1$ is an inertia weight, c_1, c_2= 1 are constants, r_1, $r_2$$\sim$ $\mathcal{U}(0,1)$ are uniform distributed randoms and p_i and p_g are the positions for the optimum of particle i and the global optimum of the swarm, respectively. x_i is the position of particle i.