Hardware as Policy: Mechanical and Computational Co-Optimization using Deep Reinforcement Learning

Anonymous Author(s)
Affiliation
Address
email

Abstract: Deep Reinforcement Learning (RL) has shown great success in learning complex control policies for a variety of applications in robotics. However, in most such cases, the hardware of the robot has been considered immutable, modeled as part of the environment. In this study, we explore the problem of learning hardware and control parameters together in a unified RL framework. To achieve this, we propose to model aspects of the robot's hardware as a "mechanical policy", analogous to and optimized jointly with its computational counterpart. We show that, by modeling such mechanical policies as auto-differentiable computational graphs, the ensuing optimization problem can be solved efficiently by gradient-based algorithms from the Policy Optimization family. We present two such design examples: a toy mass-spring problem, and a real-world problem of designing an underactuated hand. We compare our method against traditional co-optimization approaches, and also demonstrate its effectiveness by building a physical prototype based on the learned hardware parameters.

Keywords: Mechanical-Computational Co-Optimization, Reinforcement Learning

1 Introduction

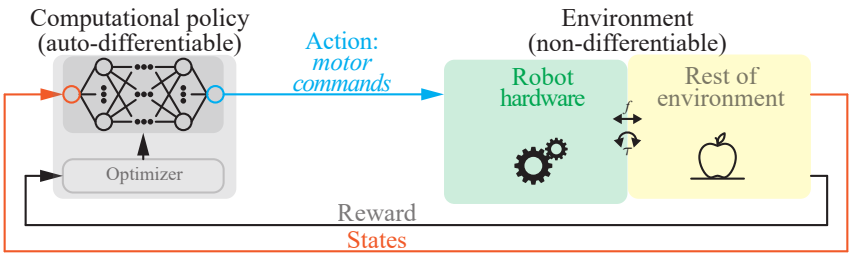
Human "intelligence" resides in both the brain and the body: we can develop complex motor skills, and the mechanical properties of our bones and muscles are also adapted to our daily tasks. Numerous motor skills exhibit this phenomenon, from running (where the stiffness of the Achilles tendon has been shown to maximize locomotion efficiency [1]) to grasping (where coordination patterns between finger joints emerge from both synergistic muscle control and mechanical coupling of joints [2]). Mechanical adaptation and motor skill improvement can happen simultaneously, both over an individual's lifetime (e.g. [3]) and at evolutionary time scales. For example, it has been suggested that, as early hominids practiced throwing and clubbing, hand morphology also changed accordingly, as the thumb got longer to provide better opposition [4].

In robotics, the idea of jointly designing/optimizing the mechanical and computational aspects has a long track record with remarkable advances, exploiting the fact that the morphology, transmissions, and control policies are tightly connected by the laws of physics and co-determine the robot behavior. If the policy and dynamics can be modeled analytically, traditional optimization can derive the desired values for hardware and policy parameters. When such an approach is not feasible (for example due to complex policies or dynamics), evolutionary computation has been used instead. However, these methods still have difficulty learning sophisticated motor skills in complex environments (e.g. partially observable states, dynamics with transient contacts), or are sample-inefficient in such cases.

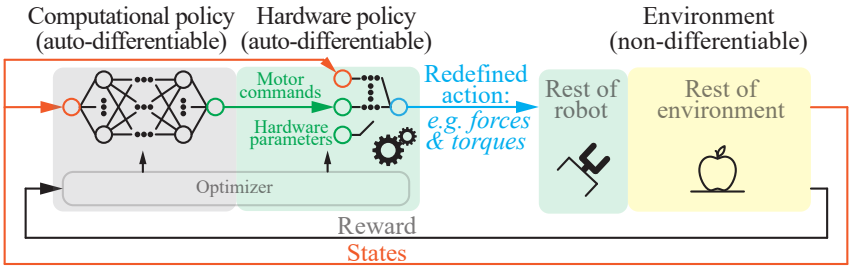
In contrast, recent advances in Deep Reinforcement Learning (Deep RL) have shown great potentials for learning difficult motor skills despite having only partial information of complex, unstructured environments (e.g. [5, 6, 7]). Traditionally, the output of a Deep RL policy in robotics consists of motor commands, and the robot hardware converts these motor commands to effects on the external world (usually through forces and/or torques). In this conventional RL perspective, robot hardware is considered given and immutable, essentially treated as part of the environment (Fig 1a).

Consider the concrete example of an underactuated robot hand. Motor forces are converted into joint torques by a transmission mechanism, consisting of gears, tendons or linkages. Through careful

Submitted to the 4th Conference on Robot Learning (CoRL 2020). Do not distribute.



(a) Reinforcement Learning with a purely computational policy



(b) Hardware as Policy-Computational Graph Implementation

Figure 1: Hardware as Policy overview. From the traditional perspective (a), all robot hardware is part of the simulated environment. In the proposed method (b), aspects of robot hardware are formulated as a "hardware policy" implemented as a computational graph, then optimized jointly with the computational policy.

design and optimization of the hardware parameters, such a transmission can provide compliant underactuation, greatly increasing the ability of the hand to grasp a wide range of objects (e.g. [8, 9]). Such a transmission is conceptually akin to a policy, mapping an input (motor forces) to an output (joint torques) with carefully tuned parameters leading to beneficial effects for overall performance.

Can we leverage the power of Deep RL for co-optimization of the computational and mechanical components of a robot? Effective sim-to-real transfer, where a policy is trained on a physics simulator and only then deployed on real hardware [10] provides such an opportunity, since it allows modifications of design parameters during training without incurring the prohibitive cost of re-building hardware. In such a case, a straightforward option is to treat these hardware parameters as hyperparameters of the RL algorithm, and optimize them via hyperparameter tuning. However, this approach carries a prohibitive computational cost.

In this study, we propose an approach to consider *hardware as policy*, optimized jointly with the traditional computational policy. As is well known, a model-free Policy Optimization (e.g. [11, 12]) or Actor-critic (e.g. [13]) algorithm can train using an auto-differentiable agent/policy and a non-differentiable black-box environment. The core idea we propose is to move some part of the robot hardware from the non-differentiable environment and into the auto-differentiable agent/policy (Fig 1b). In this way, the hardware parameters[1] become parameters in the policy graph, analogous to and optimized in the same fashion as the neural network weights and biases. Therefore, the optimization of hardware parameters can be directly incorporated into the existing RL framework, and can use existing learning algorithms with minor changes only in the computational graphs. We summarize our major contribution as follows:

- To the best of our knowledge, we are the first to express hardware aspects as a policy, in a way that allows an optimization algorithm to include gradients of actions w.r.t. hardware parameters and computational parameters.
- Via case studies comprising both a toy problem and a real-world design challenge, we show that such gradient-based methods are superior to hyperparameter tuning as well as gradient-free evolutionary strategies for hardware-software co-optimization.
- To the best of our knowledge, we are the first to build a physical prototype to validate a Deep RL-based co-optimization approach, in the form of a compliant underactuated robot hand.

[1]This paper primarily focuses on the mechanical aspect of hardware, we use the terms "hardware" and "mechanics/mechanical" interchangeably. However, we believe that, in the future, the proposed idea could be extended to electrical or sensorial aspects of a physical device.

2 Related Work

The first category of related work comprises studies using analytical dynamics and classical control. An early example is from Park and Asada [14]. Paul and Bongard [15], Geijtenbeek et al. [16] and Ha et al. [17] performed optimizations of mechanical and control or planning parameters for legged locomotors. All studies above require an analytical model of the complete mechanical-control system, which is non-trivial in complex problems. More recent work that uses classical control but evaluates and iterates on real hardware is [18], which optimizes micro robots with Baysian Optimization. However, the goal of this work is different from ours: it aims to drastically decrease the number of real-world design evaluations, which is avoided in our work by simulation and sim-to-real transfer.

Evolutionary computation provides another way to approach this problem. This research path originated from studies on the evolution of artificial creatures [19], where the morphology and the neural systems are both encoded as graphs and generated using genetic algorithms. Lipson and Pollack [20] introduced the automatic lifeform design technique using bars, joints, and actuators as building blocks of the morphology, with neurons attached to them as controllers. A series of works from Cheney et al. [21, 22] studied the morphology-computation co-evolution of cellular automata, in the context of locomotion. Nygaard et al. [23] presented a method that optimizes the morphology and control of quadruped robot using real-world evaluation of the robot. Evolutionary strategies, which are gradient-free, have significant promise, but also exhibit high computational complexity and data-inefficiency compared to recent gradient-based optimization methods.

The recent influx of reinforcement learning provides a new perspective on this co-optimization problem. Ha [24] augmented the REINFORCE algorithm with rewards calculated using the mechanical parameters. Schaff et al. [25] proposed a joint learning method to construct and control the agent, which models both design and control in a stochastic fashion and optimizes them via a variation of Proximal Policy Optimization (PPO). Vermeer et al. [26] showed a study on two-dimensional linkage mechanism synthesis using a Decision-Tree-based mechanism representation fused with Reinforcement Learning. Luck et al. [27] presented a method for data-efficient co-adaptation of morphology and behaviors based on Soft Actor-Critic (SAC), leveraging previously tested morphology and behaviors to estimate the performance of new candidates. In all the studies above, hardware parameters are still optimized separately and iteratively with the computational policies, whereas we aim to optimize both together in a unified framework. In addition, none of these works show physical prototypes based on the co-optimized agent.

Recent work on general-purpose auto-differentiable physics [28, 29, 30, 31] is also very relevant to our approach, which relies on modeling (part of) the robot hardware as an auto-differentiable computational graph. We hope to make use of such recent advances in general differentiable physics simulation in further iterations of our method.

3 Preliminaries

We start from a standard RL formulation, where the problem of optimizing an agent to perform a certain task can be modeled as a Markov Decision Process (MDP), represented by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where $\mathcal{S}$ is state space, $\mathcal{A}$ is the action space, $\mathcal{R}(\boldsymbol{s}, \boldsymbol{a})$ is the reward function, and $\mathcal{F}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})$ is the state transition model ($\boldsymbol{s}, \boldsymbol{s}' \in \mathcal{S}$, $\boldsymbol{s}'$ is for the next time step, $\boldsymbol{a} \in \mathcal{A}$). Behavior is determines by a computational control policy $\pi_{\boldsymbol{\theta}}^{comp}(\boldsymbol{a}|\boldsymbol{s})$, where $\boldsymbol{\theta}$ represents the parameters of the policy. Usually, $\pi_{\boldsymbol{\theta}}^{comp}$ is represented as a deep neural network, with $\boldsymbol{\theta}$ consisting of the network's weights and biases. The goal of learning is to find the values of the policy parameters that maximize the expected return $\mathbb{E}[\sum_{t=0}^{T} \mathcal{R}(\boldsymbol{s}_t, \boldsymbol{a}_t)]$ where T is the length of episode.

We start from the observation that, in robotics, in addition to the parameters $\boldsymbol{\theta}$ of the computational policy, the design parameters of the hardware itself, denoted here by $\boldsymbol{\phi}$, play an equally important role for task outcomes. In particular, hardware parameters $\boldsymbol{\phi}$ help determine the output (the effect on the outside world) that is produced by a given input to the hardware (motor commands). This is perfectly analogous to computational parameters $\boldsymbol{\theta}$ help determine the output of the computational policy (action $\boldsymbol{a}$) that is produced by a given input (state or observations $\boldsymbol{s}$).

Even though this analogy exists, traditionally, these two classes of parameters have been treated very differently in RL: computational parameters can be optimized via gradient-based methods — taking Policy Optimization (e.g. Trust Region Policy Optimization (TRPO) [11] and Proximal Policy Optimization (PPO) [12]) as an example learning algorithm, the parameters

of the computational policy are optimized by computing and following the policy gradient: $\boldsymbol{g} = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}^{comp}}[\sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}^{comp}(\boldsymbol{a}_t|\boldsymbol{s}_t) A_t(\boldsymbol{s}_t, \boldsymbol{a}_t)]$, where A_t is the advantage function. In contrast, hardware is generally considered immutable, and modeled as part of the environment. Formally, this means that hardware parameters $\boldsymbol{\phi}$ are considered as parameters of the transition function $\mathcal{F} = \mathcal{F}_{\boldsymbol{\phi}}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})$ instead of the policy. This is the concept illustrated in Fig. 1a. Such a formulation is grounded in the most general RL framework, where $\mathcal{F}$ is not modeled analytically, but only observed by execution on real hardware. In such a case, changing $\boldsymbol{\phi}$ can only be done by building a new prototype, which is generally impractical.

However, in recent years, the robotics community has made great advances in training via a computational model of the transition function $\mathcal{F}$, often referred to as a physics simulator (e.g. [32]). The main drivers have been the need to train using many more samples than possible with real hardware, and ensure safety during training. Recent results have indeed shown that it is often possible to train exclusively using an imperfect analytical model of $\mathcal{F}$, and then transfer to the real world [10].

In our context, training with such physics simulator opens new possibilities for hardware design: we can change the hardware parameters $\boldsymbol{\phi}$ and test different hardware configurations on-the-fly inside the simulator, without incurring the cost of re-building a prototype.

4 Hardware as Policy

The Hardware as Policy method (HWasP) proposed here largely aims to perform a similar optimization for hardware parameters as we do for computational policy parameters, i.e. by computing and follow the gradient of action probabilities w.r.t such parameters.

The core of the HWasP method is to model the effects of the robot hardware we aim to optimize separately from the rest of the environment. We refer to this component as a "hardware policy", and denote it via $\pi_{\boldsymbol{\phi}}^{hw}(\boldsymbol{a}^{new}|\boldsymbol{s}, \boldsymbol{a})$. The input to the hardware policy consists of the action produced by the computational policy (i.e. a motor command) and other components of the state; the output is in a redefined action space $\mathcal{A}^{new}$ further discussed below.

In the traditional formulation outlined so far, the "hardware policy" and its parameters $\boldsymbol{\phi}$ are included in the transition distribution function $\mathcal{F}_{\boldsymbol{\phi}}$. With HWasP, $\pi_{\boldsymbol{\phi}}^{hw}$ becomes part of the agent. The new overall policy $\pi_{\boldsymbol{\theta},\boldsymbol{\phi}} = \pi_{\boldsymbol{\phi}}^{hw}(\boldsymbol{a}^{new}|\boldsymbol{s}, \boldsymbol{a}) \pi_{\boldsymbol{\theta}}^{comp}(\boldsymbol{a}|\boldsymbol{s})$ comprises the composition of both computational and mechanical policies, while the new transition probability $\mathcal{F}^{new} = \mathcal{F}^{new}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a}^{new})$ encapsulates the rest of the environment. In other words, we have split the simulation of the environment into two: one part consists of the mechanical policy, now considered as part of the agent, while the other simulates all other components of the robot, as well as the external environment. The reward function, $\mathcal{R}(\boldsymbol{s}, \boldsymbol{a})$, will be redefined to be associated with the new action space: $\mathcal{R}^{new}(\boldsymbol{s}, \boldsymbol{a}^{new})$. Once this modification is performed, we aim to run the original Policy Optimization algorithm on the new tuple $(\mathcal{S}, \mathcal{A}^{new}, \mathcal{F}^{new}, \mathcal{R}^{new})$ as redefined above. However, in order for this to be feasible, two key conditions have to be met:

Condition 1: The redefined action vector $\boldsymbol{a}^{new}$ must encapsulate the interactions between the mechanical policy, and the rest of the environment. In other words, this new action interface must comprise all the ways in which the hardware we are optimizing effects change on the rest of the environment. Furthermore, the redefined action vector must be low-dimensional enough to allow for efficient optimization. Such an interface is problem-specific. Forces / torques between the robot and the environment make good candidates, as we will exemplify in the following sections.

Condition 2: To use Policy Optimization algorithms, we need to efficiently compute the gradient of the redefined action probability w.r.t. hardware parameters. We further discuss this condition next.

Computational Graph Implementation (HWasP). In order to meet Condition 2 above, *we propose to simulate the part of hardware we care to optimize as a computational graph*. In this way, the gradients can be computed by auto-differentiation and can flow or back-propagate through the hardware policy. Similar to the computational policy, the gradient of log-likelihood of actions w.r.t mechanical parameters $\boldsymbol{\phi}$ can be computed as $\nabla_{\boldsymbol{\phi}} \log \pi_{\boldsymbol{\phi}}^{hw}(\boldsymbol{a}^{new}|\boldsymbol{a}, \boldsymbol{s})$.

Critically, since the computational policy is also generally expressed as a computational graph, the gradient can back-propagate through both the hardware policy and the computational policy, i.e., the hardware and computational parameters are optimized jointly, and in the same fashion. This general idea is illustrated in Fig. 1b.

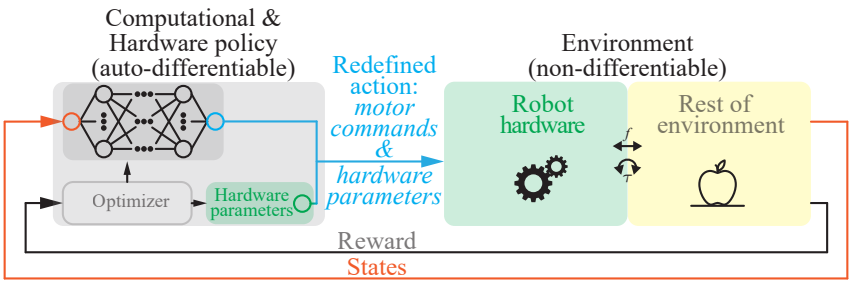


Figure 2: Hardware as Policy-Minimal

However, this approach is predicated on being able to simulate the effects of the hardware being optimized as a computational graph. Once again, the exact form of this simulation is problem-specific, and can be considered as a key part of the algorithm. In the next sections, we illustrate how this can be done both for a toy problem, and for a real-world design problem, and regard these implementations as an intrinsic part of the contribution of this work.

Minimal Implementation (HWasP-Minimal). In the general case, where should the split between the (differentiable) hardware policy and the (non-differentiable) rest of the environment simulation be performed? In particular, what if the hardware we care to optimize does not lend itself to a differentiable simulation using existing methods?

Even in such a case, we argue that a "minimal" hardware policy is always possible: we can simply put the hardware parameters into the output layer of the original computational policy. In this case, $\boldsymbol{a}^{new} = [\boldsymbol{a}, \boldsymbol{\phi}]^T$. Here, the policy gradient with respect to the hardware parameters is trivial but can be still useful to guide the update of parameters. When this case in implemented in practice, the transition function $\mathcal{F}(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a}^{new})$ typically operates in two steps: first, it sets the new values of the hardware parameters to the underlying simulator, then advances the simulation to the next step.

We illustrate this case in Fig. 2, which can be directly compared to the general HWasP in Fig. 1b. HWasP-Minimal is simple to implement since it does not require a physics-based auto-differentiable hardware policy. As outlined in the following sections, this version still performs at least as well as or better than our baselines, but still below HWasP.

Comparison Baselines. We compare HWasP and HWasP-Minimal with the following baselines:

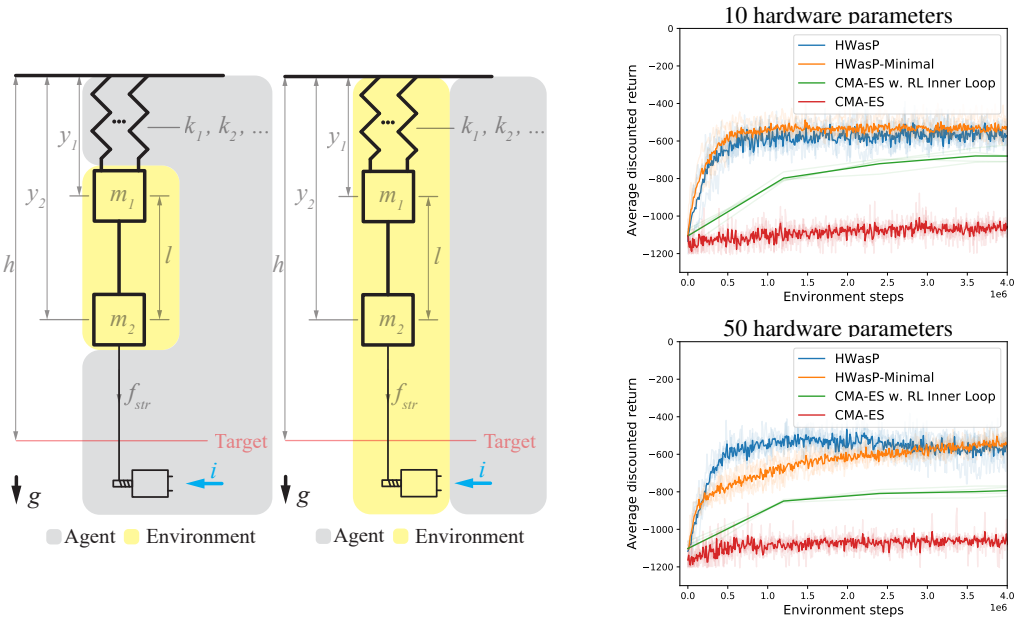
- CMA-ES with RL inner loop: here, we treat hardware parameters as hyperparameters, using the Covariance Matrix Adaptation - Evolution Strategy (CMA-ES) algorithm [33] in an outer loop that optimizes hardware parameters, while learning the policy using RL algorithms (e.g. PPO or TRPO) in an inner loop for each set of sampled hardware parameters.
- CMA-ES: here, we use CMA-ES as a gradient-free evolutionary strategy to directly learn both computational policy and hardware parameters, without a separate inner loop.

5 A Mass-spring Toy Problem

We present here a simple one-dimensional implementation of our method on the mass-spring system in Fig.3(a). Two point masses, connected by a massless bar, are hanging in the standard gravity field under n parallel springs whose stiffnesses are $k_1, \ldots, k_n$. A motor can apply a controllable force to the lower mass. The behavior is governed by a computational policy that regulates the motor force, but also by the hardware parameters (spring stiffnesses). We note that, since springs are all parallel, only the sum of their stiffnesses matters, but we still consider the stiffness of each individual spring as a parameter as a way to test how our methods scale up for higher-dimensional problems.

The input to the computational policy consists of y_2 and $\dot{y}_2$, and the output of the computational policy is motor current i. The goal is to optimize both the computational policy that regulates motor force, and the hardware parameters such that the lower mass goes to the red target line ($y_2 = h$) and stay there, with minimum motor effort. (The exact formulation for the reward function we use is presented in the Supplementary Materials.)

Hardware as Policy. In this case, we include the effect of the parallel springs in the mechanical policy. Using Hooke's Law, we model spring effects as a simple computational graph, with $k_1, \ldots, k_n$ as parameters. The output of this computational graph is the total spring force f_{spr} applied to the masses.



(a) Problem description. The agent-environment split is shown for HWasP (left) and HWasP-Minimal (right) implementations.

(b) Learning curves for the mass-spring toy problem.

Figure 3: A mass-spring system and corresponding learning curves.

The redefined action $\boldsymbol{a}^{new}$ consists of the total resultant force $f_{total} = f_{str} + f_{spr}$. The transition function $\mathcal{F}$ (rest of the environment) implements Newton's Law for the two masses, assuming f_{total} as an external force. Additional details for the implementation, including the structure of the computational graph, can be found in Supplementary Materials.

Hardware as Policy — Minimal. Here, we simply re-define the action vector to also include spring stiffnesses: $\boldsymbol{a}^{new} = [i, k_1, \cdots, k_n]^T$. The transition function $\mathcal{F}$ is responsible for modeling the dynamics of the springs and the two masses.

Results. Fig.3(b) shows the comparison of the training curves for both implementations of our method, as well as other baselines, for two cases: one with 10 parallel springs, and one with 50 parallel springs. In both cases, HWasP learns an effective joint policy that moves the lower mass to the target position. HWasP-Minimal works equally well for the smaller problem, but suffers a drop in performance as the number of hardware parameters increases. CMA-ES with RL inner loop also learns a joint policy, but learns slower than our method, especially for the larger problem. CMA-ES by itself does not exhibit any learning behavior over the number of samples tested. For the numerical results of the optimized stiffnesses, please refer to the Supplementary Materials.

6 Co-Design of an Underactuated Hand

In this section we show how HWasP can be applied to a real-world design problem: optimizing both the mechanism and the control policy for an underactuated robot hand. The high-level goal of this problem, similar to the one introduced by Chen et al. [34] and illustrated in Fig 4, is to design a robot hand that is simultaneously versatile (able to grasp different shaped objects) and compact.

In order to achieve the stated compactness goal, all joints are driven by a single motor, via an underactuated transmission mechanism: A single tendon connects to the motor, then splits to actuate all joints in the flexion direction (see Fig 4). Finger extension is passive, provided by preloaded torsional springs. The mechanical parameters that govern the behavior of this mechanism consist of tendon pulley radii in each joint, as well as stiffness values and preload angles for restoring springs.

Here, we look to simultaneously optimize the hardware parameters along with a computational policy that determines how to position the hand, and when to use the motor. The input to the computational policy consists of the position vectors of the palm and object, the size vector of the object bounding-box, the current hand motor travel and motor torque. Its output contains relative motor travel and palm motion commands.

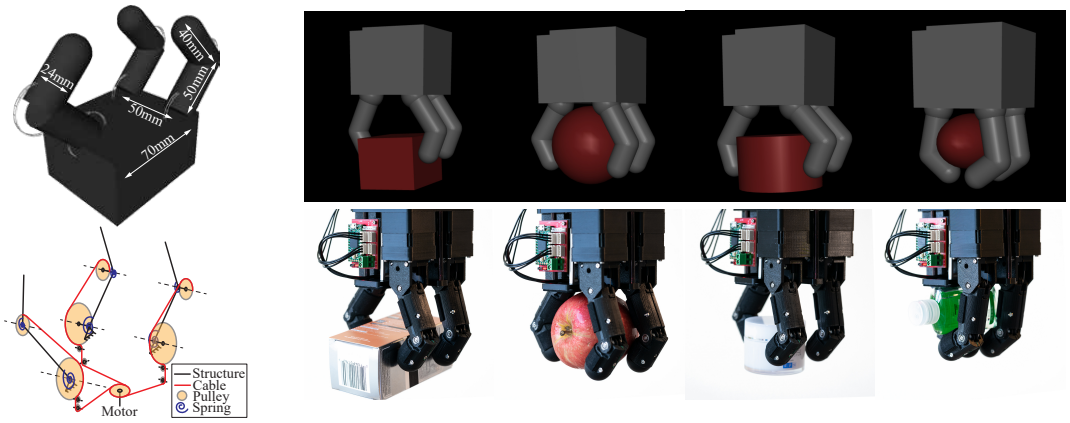


Figure 4: Hand design optimization problem. Left: hand kinematics, dimension, and tendon routing. Right: successful grasps executed in simulation and on a real hand prototype.

The hardware parameters we aim to optimize consist of all parameters of the underactuated transmission as listed above. It is important to note that, in this study, we do not try to optimize the kinematic structure or topology for the hand. Unlike the underactuated transmission, these aspects do not lend themselves to parameterization and implementation as computational graphs, preventing the use of the HWasP method in its current form. While HWasP-Minimal could still be applied, we leave that for future investigations.

We tested our method with two grasping tasks: 1. top-down grasping with only z-axis motion for the palm movement (Z-Grasp); 2. top-down grasping with 3-dimensional palm motion (3D-Grasp). The former is a simplified problem version of the latter, and easier to train. Additional details on problem formulation and training can be found in Supplementary Materials.

Noted that since the hardware parameters can be large in scale comparing to weights and biases in the neural network, a small change of them can lead to a huge shift of the joint policy output distribution during training. This kind of large distribution shift can result in a local optimum in the reward landscape. Hence, we hope to improve our policy performance while having small changes of the joint policy output distribution. In this problem, we use TRPO [11] because it allows for hard constraints on the change of action distribution.

We also apply Domain Randomization [10] in the training to increase the chance of successful sim-to-real transfer. We randomized object shape, size, weight, friction coefficient and inertia, injected sensor and actuation noise, and applied random disturbance wrenches on the hand-object system.

Hardware as Policy. In this case, we model the complete underactuated transmission as a computational graph and include it in our mechanical policy. The input to the mechanical policy consists of the commanded motor travel (output by the computational policy), as well current joint angles. Its output consists of hand joint torques. To perform this computation, we use a tendon model that computes the elongation of the tendon in response to motor travel and joint positions, then uses that value to compute tendon forces and joint torques. Details of this model as well as its implementation as an auto-differentiable computational graph can be found in Supplemental Materials.

The redefined action $\boldsymbol{a}^{new}$ contains the palm position command output by the computational policy, and the joint torques produced by the mechanical policy. The rest of the environment comprises the hand-object system without the tendon underactuation mechanisms, i.e. with independent joints.

Hardware as Policy — Minimal. In this case, all hardware parameters are simply appended to the output of the computational policy. The underactuated transmission model is part of the environment, along with the rest of the hand as well as the object.

Results. Our results are shown in Fig. 5). In the case of the Z-Grasp problem (left), HWasP learns an effective computational/hardware policy, albeit with some measure of instability in the learning curve. HWasP-Minimal also learns, but lags in performance. Neither evolutionary strategy shows any learning behavior over a similar number of training steps.

We also tried a version of the same problem with the search range for the hardware parameters reduced by a factor of 8 (middle plot). Here, all methods except CMA-ES obtain similarly effective

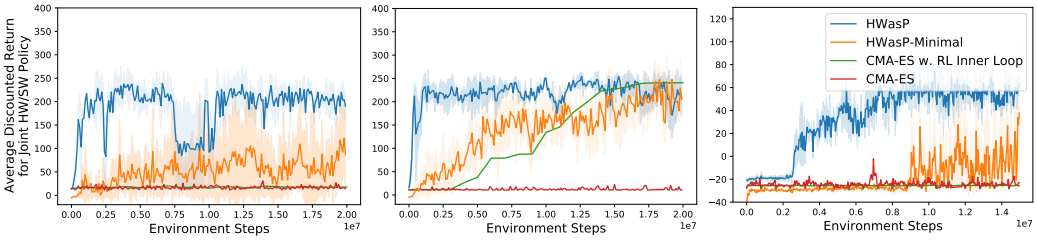


Figure 5: Training curves for the grasping problem. Left: Z-Grasp with a large hardware parameter search range. Middle: Z-Grasp with small hardware search range. Right: 3D-Grasp with a small search range.

policies, but HWasP is still most efficient. Finally, we investigated performance for the more complex 3D-Grasp task. With a large search range, neither method was able to learn. However, with a reduced search range, HWasP was able to learn an effective policy, while neither CMA-ES-based method displayed any learning behavior over a similar timescale. The values for the hardware parameters resulting from the optimizations are shown in the Supplementary Materials.

Validation with Physical Prototype. To validate our results in the real world, we physically built the hand with the parameters resulted from the co-optimization. The hand is 3D printed, and actuated by a single position-controlled servo motor. Fig. 4 shows some grasps obtained by this physical prototype, compared to their simulated counterparts. We note that, by virtue of a large number of simulation samples of different grasp types with different object shapes, sizes and other physics properties, the hand is versatile and can perform both stable fingertip grasps as well as enveloping grasps for different objects in reality.

7 Disscussion and Conclusion

Our results show that the HWasP approach is able to learn combined computational and mechanical policies. We attribute this performance to the fact that HWasP connects different hardware parameters via a computational graph based on the laws of physics, and can provide the physics-based gradient of the action probability w.r.t the hardware parameters. The HWasP-Minimal implementation does not provide such information, and the policy gradient can only be estimated via sampling, which is usually less efficient, particularly for high-dimensional problems. In consequence, HWasP-Minimal also shows the ability to learn effective policies, but with reduced performance.

Compared to gradient-free evolutionary baselines for joint hardware-software co-optimization, HWasP always learns faster, while HWasP-Minimal is at least as effective as the best baseline algorithm. We note that combining an RL inner loop for the computational policy with a CMA-ES outer loop for hardware parameters proved more effective than directly using CMA-ES for the complete problem. Still, HWasP outperforms both methods.

The biggest advantage of HWasP-Minimal is that, like gradient-free methods, it does not depend on auto-differentiable physics, and is widely applicable with straightforward implementations to various problems using existing non-differentiable physics engines. We believe that our methods represent a step towards a framework where an algorithm designer can ”tune the slider” to decide how much physics to include in the computational policy, based on the trade-offs between computation efficiency, ease of development, and the availability of auto-differentiable physics simulations.

In its current stage, our work still presents a number of limitations. In particular, HWasP suffers from stability issues when the parameter search range is large. We suspect that this is due to the relative scale of the hardware parameters (imposed by the laws of physics), which can be large enough to scale the gradient through the hardware computational graph and create instability. Partly due to this problem, the computational aspects of the policies we have explored so far are relatively simple (e.g. limiting hand motion to 1- or 3-DOF). We hope to explore more challenging robotic tasks in future work, for example 6-DOF grasping problems. Finally, we also aim to include additional hardware aspects in the optimization, such as mechanism kinematics, morphology, or link dimensions.

We believe the proposed idea of considering hardware as part of the policy will enable us to co-design of hardware and software using existing RL toolkits, with changes in the computational graph structure but no changes in the learning algorithms. We hope this work can open up new opportunities for task-based hardware-software co-design of robots or other intelligent systems, for researchers both in RL and in the hardware domain.

References

[1] G. Lichtwark and A. Wilson. Is achilles tendon compliance optimised for maximum muscle efficiency during locomotion? *Journal of biomechanics*, 40(8):1768–1775, 2007.

[2] M. Santello, G. Baud-Bovy, and H. Jörntell. Neural bases of hand synergies. *Frontiers in computational neuroscience*, 7:23, 2013.

[3] R. Hammami, A. Chaouachi, I. Makhlouf, U. Granacher, and D. G. Behm. Associations between balance and muscle strength, power performance in male youth athletes of different maturity status. *Pediatric Exercise Science*, 28(4):521–534, 2016.

[4] R. W. Young. Evolution of the human hand: the role of throwing and clubbing. *Journal of Anatomy*, 202(1):165–174, 2003.

[5] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, and S. Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*, 2017.

[6] M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, et al. Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*, 2018.

[7] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine. Learning to walk via deep reinforcement learning. *arXiv preprint arXiv:1812.11103*, 2018.

[8] L. Birglen and C. M. Gosselin. Kinetostatic analysis of underactuated fingers. *IEEE Transactions on Robotics and Automation*, 20(2):211–221, 2004.

[9] L. U. Odhner, L. P. Jentoft, M. R. Claffee, N. Corson, Y. Tenzer, R. R. Ma, M. Buehler, R. Kohout, R. D. Howe, and A. M. Dollar. A compliant, underactuated hand for robust manipulation. *The International Journal of Robotics Research*, 33(5):736–752, 2014.

[10] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017.

[11] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *Intl. Conf. on machine learning*, pages 1889–1897, 2015.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[13] 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.

[14] J.-H. Park and H. Asada. Concurrent design optimization of mechanical structure and control for high speed robots. *Journal of dynamic systems, measurement, and control*, 116(3):344–356, 1994.

[15] C. Paul and J. C. Bongard. The road less travelled: Morphology in the optimization of biped robot locomotion. In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No. 01CH37180)*, volume 1, pages 226–232. IEEE.

[16] T. Geijtenbeek, M. Van De Panne, and A. F. Van Der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics (TOG)*, 32(6):206, 2013.

[17] S. Ha, S. Coros, A. Alspach, J. Kim, and K. Yamane. Computational co-optimization of design parameters and motion trajectories for robotic systems. *The International Journal of Robotics Research*, 37(13-14):1521–1536, 2018.

[18] T. Liao, G. Wang, B. Yang, R. Lee, K. Pister, S. Levine, and R. Calandra. Data-efficient learning of morphology and controller for a microrobot. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 2488–2494. IEEE, 2019.

[19] K. Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM, 1994.

[20] H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406(6799):974, 2000.

[21] N. Cheney, R. MacCurdy, J. Clune, and H. Lipson. Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding. *ACM SIGEVOlution*, 7(1):11–23, 2014.

[22] N. Cheney and H. Lipson. Topological evolution for embodied cellular automata. *Theoretical Computer Science*, 633:19–27, 2016.

[23] T. F. Nygaard, C. P. Martin, E. Samuelsen, J. Torresen, and K. Glette. Real-world evolution adapts robot morphology and control to hardware limitations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 125–132, 2018.

[24] D. Ha. Reinforcement learning for improving agent design. *arXiv preprint arXiv:1810.03779*, 2018.

[25] C. Schaff, D. Yunis, A. Chakrabarti, and M. R. Walter. Jointly learning to construct and control agents using deep reinforcement learning. In *IEEE Intl. Conf. on Robotics and Automation*, pages 9798–9805. IEEE, 2019.

[26] K. Vermeer, R. Kuppens, and J. Herder. Kinematic synthesis using reinforcement learning. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 51753, page V02AT03A009. ASME, 2018.

[27] K. S. Luck, H. Ben Amor, and R. Calandra. Data-efficient co-adaptation of morphology and behaviour with deep reinforcement learning. In *Conf. on Robot Learning*, 2019.

[28] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, pages 7178–7189, 2018.

[29] J. Degrave, M. Hermans, J. Dambre, and F. Wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers in neurorobotics*, 13:6, 2019.

[30] Y. Hu, J. Liu, A. Spielberg, J. B. Tenenbaum, W. T. Freeman, J. Wu, D. Rus, and W. Matusik. Chainqueen: A real-time differentiable physical simulator for soft robotics. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6265–6271. IEEE, 2019.

[31] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.

[32] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

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

[34] T. Chen, L. Wang, M. Haas-Heger, and M. Ciocarlie. Underactuation design for tendon-driven hands via optimization of mechanically realizable manifolds in posture and torque spaces. *IEEE Transactions on Robotics*, 2020.