RMP^2: A Differentiable Policy Class for Robotic Systems with Control-Theoretic Guarantees

Anqi Li[1]*, **Ching-An Cheng**[2]*, **M. Asif Rana**[3], **Nathan Ratliff**[4], **and Byron Boots**[1,4]
[1]University of Washington, [2]Microsoft Research, [3]Georgia Institute of Technology, [4]NVIDIA
anqil4@cs.washington.edu

Abstract

We consider the problem of achieving safe robot learning through learning on an inherently safe policy class. The safe policy class is inspired by RMPflow, a framework for robot policy design that provides control-theoretic guarantees. We re-examine the computational structure of RMPflow, and propose a more efficient alternate algorithm, called RMP^2 (RMPflow Reactive Motion Policy), that uses modern automatic differentiation tools (such as TensorFlow and PyTorch) to compute the RMPflow policy. Our reduction retains the strengths of RMPflow while bringing in advantages from automatic differentiation, including *1)* support of general directed acyclic graph (DAG) transformation structures; *2)* easy programming interfaces to designing complex transformations; *3)* improved computational efficiency; *4)* end-to-end differentiability for policy learning. Because of these features, RMP^2 can be treated as a differentiable policy class for learning with control-theoretic guarantees, which is suitable for safe exploration and encoding domain knowledge. Our experiment gives a proof-of-concept demonstration on a reinforcement learning problem for goal reaching in a cluttered space.

1 Introduction

Reinforcement learning algorithms often rely on random explorations to collect information about costs or rewards in the environment [1]. Although this strategy has been effective in simulated tasks, for robotic systems, random explorations can present a significant danger to humans, the environment, and the robot itself. For such systems, safe learning is important: not only shall the final learned policies be safe, *any intermediate policies* that get executed on the robot during learning should also have safety guarantees. Safe reinforcement learning has mainly been addressed in two ways: *1)* through safe policy improvement algorithms and *2)* through safe policy classes. Safe policy improvement algorithms focuses on policy updates that result in bounded constraint violation during learning [1, 20, 22], while the other methodology provides safety guarantees by directly learning on an inherently safe policy class through, e.g. projected policy gradient [5, 6, 8]. Our work is in line with the latter: we propose a differentiable policy class for robot learning with control-theoretic guarantees, which is inspired by RMPflow [3], a recently introduced framework for reactive motion control that can perform stable and safe policy fusion with computational efficiency.

In RMPflow [3], the full robot control problem is decomposed into smaller subtasks, such as goal reaching, collision avoidance, maintaining balance, etc., each of which is achieved by a reactive policy. RMPflow treats these task spaces, where the task specifications are described, as manifolds mapped from the configuration space of the robot, and defines motion policies to satisfy a certain geometric consistency induced by the geometries of these task spaces. In [3], it is proved that the policy RMPflow constructed is stable and can be efficiently computed through *message passing* on a tree-structured task map. Because of the control-theoretic guarantees and computation efficiency, RMPflow has been applied to many robotic applications [11–14, 16, 21, 23].

Policy learning with the structure of RMPflow preserve several advantages: *1*) It allows the policies to partially parameterized. As an example, one can hand design safety-critical subtask policies for, e.g.

* Equal contribution

NeurIPS 2020 3rd Robot Learning Workshop: Grounding Machine Learning Development in the Real World.

collision avoidance, and learn others to improve the policy performance; *2*) As a result, it provides non-statistical guarantees, such as stability and safety, throughout learning; *3*) The policy learned on one robot can be transferred to other robots and tasks [2]; and finally, *4*) The policy parameterization is shown to be expressive in generating complex robot motion with hand-designed subtask policies [3], and these hand-designed substask policies can be used to partially parameterize the policy.

However, the RMPflow algorithm was originally designed for reactive control rather than learning: *following the original RMPflow message passing algorithm can risk of undermining the flexibility and computational efficiency in learning*. One reason is that RMPflow uses a rather complicated user interface involving a tree data structure that is non-trivial for users to specify. The other reason, which is perhaps more important, is that it is hard and inefficient to trace the gradient flow in the RMPflow message passing algorithm. Although recent work has looked into learning with RMPflow [14, 15, 17], these methods either largely simplify the learning problem so that differentiating through RMPflow is not needed [14, 17] or only allows for a very limited parameterization of the policy [15]. For example, Meng et al. [14] and Rana et al. [17] learn the subtask policies independently through imitation and then use RMPflow to combine the learned RMPs with other hand-specified policies; and Mukadam et al. [15] learn scalar weight functions for pre-defined subtask polices.

In this work, we propose a simple alternative algorithm, called RMP^2, to realize the policy output by RMPflow for end-to-end learning, *without* using the tree data structure and message passing steps of the RMPflow algorithm [3]. Instead RMP^2 operates by querying the Gradient Oracle (though back-propagation) in automatic differentiation [9]. RMP^2 has several advantage over the original RMPflow algorithm: *1*) RMP^2 relaxes the assumption on using tree-structured task maps in the RMPflow algorithm [3] to work with *any* directed acyclic graph (DAG) task maps. *2*) RMP^2 allows for a simper user interface. The user only needs to specify the task map using an automatic differentiation library, the automatically constructed (directed acyclic) computational graph can then be used for computing the RMPflow policy. *3*) RMP^2 uses a smaller memory footprint than RMPflow, while having the same time complexity. *4*) RMP^2 is much easier to implement in conjunction with learning algorithms: as RMP^2 is implemented using operators supported by automatic differentiation libraries, it is convenient to take the gradient, or higher order derivatives, of *any function* with respect to the parameters in subtask policies and task maps. This makes RMP^2 a policy class that is generally applicable to many end-to-end learning scenarios and algorithms.

2 Background

2.1 Setup: Configuration Space and Task Space

Suppose the robot's configuration space $\mathcal{C}$ (e.g. the joint space of a robot with revolute joints) is a d-dimensional smooth manifold that can be described by generalized coordinates $\mathbf{q} \in \mathbb{R}^d$. For simplicity of exposition, we assume that the acceleration of the robot $\ddot{\mathbf{q}}$ can be controlled directly. This condition is true when the robot is fully actuated and has been feedback linearized by, e.g., using an inverse dynamics model. We will call $(\mathbf{q}, \dot{\mathbf{q}})$ the state of the robot on the configuration space $\mathcal{C}$, as we focus on acceleration-controlled systems.

In motion control problems, typically a task is not directly described in the generalized coordinates $\mathbf{q} \in \mathbb{R}^d$, but in terms of another set of task coordinates $\mathbf{x} \in \mathbb{R}^m$ that are related to the generalized coordinates through a nonlinear mapping ψ, i.e. $\mathbf{x} = \psi(\mathbf{q})$. We call this mapping ψ the *task map* and refer to the image manifold of $\mathcal{C}$ under ψ as the *task space*, which is denoted as $\mathcal{T}$. Our goal is to design an *acceleration* policy π such that the system following $\ddot{\mathbf{q}} = \pi(\mathbf{q}, \dot{\mathbf{q}})$ would exhibit desired behaviors when viewed in the task space $\mathcal{T}$. When the robot needs to satisfy various performance criteria in a multi-task control problem, the task space $\mathcal{T}$ can then become a complicated manifold $\mathcal{T} = \prod_{k=1}^{K} \mathcal{T}_k$ embedded in a high-dimensional ambient space, where K is the number of *subtasks* and $\mathcal{T}_k$ denotes *subtask spaces*, the manifold for the kth subtask. Importantly, the coordinates of these subtask spaces are not independent but intertwined together as the image of the common configuration space $\mathcal{C}$ under the task map ψ. For example, in an anthropomorphic robot, the torso's motion affects both the stability and the reachable region of the hands. As a result, when the number of tasks K is large, there may not be enough of degrees of freedom to find a configuration space policy π to exactly replicate the task policy without comprising the desired accelerations of all subtasks.

2.2 RMPflow

RMPflow [3] is a computational framework designed to address the multi-task control problems mentioned above. RMPflow resolves the conflicts between different subtasks by describing each

subtask policy (e.g. for the kth subtask) as a Riemannian Motion Policy (RMP) [18]: each subtask is associated not only with the desired acceleration $\mathbf{a}_k^d(\mathbf{x}, \dot{\mathbf{x}})$, but also with a positive semi-definite matrix function $\mathbf{M}_k(\mathbf{x}, \dot{\mathbf{x}})$ that depends on the state (i.e. the position and the velocity) of the subtask. Given RMPs for the subtasks, RMPflow generates the policy π on the configuration space by combining these subtask RMPs through message passing on a tree data structure. It can be proved that this final policy π is Lyapunov stable, when $\mathbf{M}_k$ is derived appropriately from a Riemannian metric that describes the motion induced by $\mathbf{a}_k^d$ for each policy [3, 12].

The RMPflow algorithm is based on two components: *1*) the RMP-tree: a directed tree encoding the structure of the task map. In the RMP-tree, a node is associated a manifold, and an edge represents a smooth map from a parent node manifold to its child node manifold. The root node of the RMP-tree and the leaf nodes correspond to the configuration space $\mathcal{C}$ and the subtask spaces $\{\mathcal{T}_k\}$ on which the RMPs are hosted, respectively. *2*) the RMP-algebra: a set of operations to propagate information across the RMP-tree. We refer the readers to [3] or Appendix A for a detailed description of RMPflow.

A connection is made in [2, Chapter 11.7] between the message passing algorithm of RMPflow and sparse linear solvers. Consider an RMP-tree with a set of nodes $\mathcal{V}$. Let $\mathcal{L} \subset \mathcal{V}$ be the set of leaf nodes and $\mathtt{r}$ be the root node. In [2], it is proved that the desired acceleration of the RMPflow policy $\pi(\mathbf{q}, \dot{\mathbf{q}})$ is the solution to the following least squares problem:

$$\begin{aligned} \min_{\{\mathbf{a}_\mathtt{v}:\mathtt{v}\in\mathcal{V}\}} \quad & \textstyle\sum_{\mathtt{v}\in\mathcal{L}} \frac{1}{2} \|\mathbf{a}_\mathtt{v} - \mathbf{a}_\mathtt{v}^d\|^2_{\mathbf{M}_\mathtt{v}}, \\ \text{s.t.} \quad & \mathbf{a}_\mathtt{v} = \mathbf{J}_\mathtt{v}\mathbf{a}_\mathtt{u} + \dot{\mathbf{J}}_\mathtt{v}\dot{\mathbf{x}}_\mathtt{u}, \quad \forall\, \mathtt{v} \in \mathcal{V}\setminus\mathtt{r}, \quad \mathtt{u} = \mathcal{P}(\mathtt{v}) \end{aligned} \tag{1}$$

where, for a leaf node $\mathtt{v} \in \mathcal{L}$, $\mathbf{a}_\mathtt{v}^d$ and $\mathbf{M}_\mathtt{v}$ together are a leaf-node RMP, $\mathbf{J}_\mathtt{v}$ denotes the Jacobian of the task map from $\mathcal{P}(\mathtt{v})$, the parent node of $\mathtt{v}$, to $\mathtt{v}$. The RMPflow algorithm is essentially an efficient routine that uses the duality of (1) and the sparsity in the task map to compute $\mathbf{a}_\mathtt{r}$. RMPflow has the time complexity of $O(Nbd^3)$ and space complexity of $O(Nd^2 + Ld^2)$, where N and L are the number of nodes and leaf nodes, b is the maximum branching factor, and d is the maximum dimension of nodes (See Appendix C for the complexity analysis).

3 RMP^2 based on Automatic Differentiation

3.1 An RMPflow Policy without the RMPflow Algorithm

If we are interested in *learning* an RMPflow policy rather than designing it, the focus changes: computational efficiency and ease of implementation when used in conjunction with learning algorithms become paramount. Machine learning pipelines commonly use automatic differentiation tools, which coincidentally provide many features that RMPflow uses, such as graph structures and the chain rule.

An important implication of the insight in [2, Chapter 11.7] (that RMPflow is a sparse linear solver) is that the RMPflow policy can also be computed by solving the unconstrained version of (1):

$$\min_{\mathbf{a}'_\mathtt{r}\in\mathbb{R}^d} \quad \textstyle\sum_{\mathtt{v}\in\mathcal{L}} \frac{1}{2} \|\mathbf{J}_{\mathtt{v};\mathtt{r}}\mathbf{a}'_\mathtt{r} + \dot{\mathbf{J}}_{\mathtt{v};\mathtt{r}}\dot{\mathbf{q}} - \mathbf{a}_\mathtt{v}^d\|^2_{\mathbf{M}_\mathtt{v}}, \tag{2}$$

where $\mathbf{J}_{\mathtt{v};\mathtt{r}}$ is the Jacobian matrix of the subtask map from the root $\mathtt{r}$ to the leaf $\mathtt{v}$. As in (1), the Jacobians and velocities here are treated as constants in the optimization as they are only dependent on the state. Note that (2) has a closed-form solution:

$$\mathbf{a}_\mathtt{r} = \underbrace{\left(\textstyle\sum_{\mathtt{v}\in\mathcal{L}} \mathbf{J}_{\mathtt{v};\mathtt{r}}^\top \mathbf{M}_\mathtt{v} \mathbf{J}_{\mathtt{v};\mathtt{r}}\right)^\dagger}_{\mathbf{M}_\mathtt{r}^\dagger} \underbrace{\left(\textstyle\sum_{\mathtt{v}\in\mathcal{L}} \mathbf{J}_{\mathtt{v};\mathtt{r}}^\top \mathbf{M}_\mathtt{v} (\mathbf{a}_\mathtt{v}^d - \dot{\mathbf{J}}_{\mathtt{v};\mathtt{r}}\dot{\mathbf{q}})\right)}_{\mathbf{f}_\mathtt{r}}. \tag{3}$$

3.2 RMP^2 based on Reverse Accumulation

We propose RMP^2 (Algorithm 1), an efficient reduction technique based on this idea. RMP^2 computes the RMPflow policy by using automatic differentiation libraries to compute the closed-form solution (3). It has the same time complexity of $O(Nbd^3)$ as RMPflow [3], while enjoying a smaller memory footprint of $O(Nd + Ld^2)$. It should be noted that explicitly calculating the Jacobian matrix $\mathbf{J}_{\mathtt{v};\mathtt{r}}$ for computing $\mathbf{M}_\mathtt{r}$ and $\mathbf{f}_\mathtt{r}$ in (3) would result in excessive time and space complexity of $O(Nbd^3L)$ and $O(NLd^2)$, respectively. To avoid this, we leverage a technique, called reverse accumulation [7] (also known as double backward), which is used to compute Jacobian-vector-product (see Algorithm 3 in Appendix) in reverse mode automatic differentiation.

In the forward pass of RMP^2 (Algorithm 1, line 3–5), the leaf node velocities $\dot{\mathbf{x}}_i = \mathbf{J}_i\mathbf{q}$ and curvature terms $\mathbf{c}_i = \dot{\mathbf{J}}_i\mathbf{q}$ are computed using Jacobian-vector-product. In the backward pass (line 7–10),

RMP^2 computes $\mathbf{M}_\mathbf{r}$ using reverse accumulation in line 10. This is accomplished by first creating a mirrored task image (line 7 and 8) and then computing $\mathbf{M}_\mathbf{r}$ by `jacobian(gradient(v,q),q')`. Similar technique is applied when computing $\mathbf{f}_\mathbf{r}$.

Algorithm 1 RMP^2

1: **Input:** root state $(\mathbf{q}, \dot{\mathbf{q}})$, task map ψ, `rmp_eval`
2: **Return:** motion policy $\pi(\mathbf{q}, \dot{\mathbf{q}})$
3: $\{\mathbf{x}_i\}_{i=1}^L \leftarrow$ `task_map(q)`
4: $\{\dot{\mathbf{x}}_i\}_{i=1}^L \leftarrow$ `jvp(`$\{\mathbf{x}_i\}_{i=1}^L$`,q,`$\dot{\mathbf{q}}$`)` *// leaf node velocity*
5: $\{\mathbf{c}_i\}_{i=1}^L \leftarrow$ `jvp(`$\{\dot{\mathbf{x}}_i\}_{i=1}^L$`,q,`$\dot{\mathbf{q}}$`)` *// curvature terms*
6: $\{(\mathbf{M}_i, \mathbf{f}_i)\}_{i=1}^L \leftarrow$ `rmp_eval(`$\{(\mathbf{x}_i, \dot{\mathbf{x}}_i)\}_{i=1}^L$`)`
7: $\mathbf{q}' =$ `copy(q)` *// create a copy and without gradient flow*
8: $\{\mathbf{x}'_i\}_{i=1}^L \leftarrow$ `task_map(q')` *// a mirrored image*
9: $\mathbf{v} \leftarrow \sum_{i=1}^L \mathbf{x}_i^\top \mathbf{M}_i \mathbf{x}'_i$, $\mathbf{w} \leftarrow \sum_{i=1}^L \mathbf{x}_i^\top (\mathbf{f}_i - \mathbf{M}_i \mathbf{c}_i)$ *// auxiliary variables*
10: $\mathbf{M}_\mathbf{r} \leftarrow$ `jacobian(gradient(v,q),q')`, $\mathbf{f}_\mathbf{r} \leftarrow$ `gradient(w,q)` *// compute root RMP*
11: $\pi(\mathbf{q}, \dot{\mathbf{q}}) \leftarrow \mathbf{M}_\mathbf{r}^\dagger \mathbf{f}_\mathbf{r}$

Since RMP^2 is implemented using the automatic differentiation libraries, computational graph can be automatically constructed while calculating the policy. This allows for convenient gradient calculation of *any function* with respect to *any parameters* in, e.g. task maps and RMPs. This fully differentiable structure is important for end-to-end learning of these parameters in many learning scenarios. Another benefit of RMP^2 is that the assumption of using tree-structured task maps can be relaxed to *arbitrary* directed acyclic graph (DAG) task maps. While Cheng et al. [3] show that every task map has a tree representation, not all motion control problems have an *intuitive* RMP-tree representation (e.g. multi-robot control [13]). If they are implemented using a tree structure, extra high-dimensional nodes would be induced and the user interface becomes tedious.

4 Experiments

We consider an RL task on a 2-d point robot as a proof of concept example for policy learning. The objective here is to reach a goal while avoiding static obstacles in the environment. The motion policy inputs include the state of the robot, the goal position, as well as the centers and radii of the obstacles. We compare reinforcement learning with the RMP^2 policy (RMPflow policy realized by RMP^2) and a feed-forward neural-net policy. The RMPflow policy is partially parameterized as the composition of a neural-net goal reaching RMP and hand-specified collision avoidance RMPs suggested in [3]. The reward given to the robot at each step is the negative Euclidean distance to the goal, with an additional penalty of -100 if the robot collides with an obstacle. We run the natural policy gradient [10] with a step size of 0.1 for 500 iterations to train the policies end-to-end to optimize the expected accumulated reward. The average accumulated reward after training for the RMP^2 policies and the neural-network policies are -37.90 and -1187.66, respectively (larger is better). Please see Appendix D for implementation details. We test the trained policies on environments both within the training distribution (Figure 1, left 2) and outside of the distribution (Figure 1, right 2). For each policy, we visualize 4 trajectories given by policies trained with different random seeds. In all the cases, the RMP^2 policy (blue) manages to reach the goal while avoiding collision with any obstacles, thanks to the additional structure provided by the hand-specified policies. The neural network policy (orange) only learns to avoid collision with obstacles. Interestingly, a common strategy among the neural network policies is to move outside the region where the obstacles are distributed during training, so that it can avoid all potential collision (and hence any chance of reaching the goal).

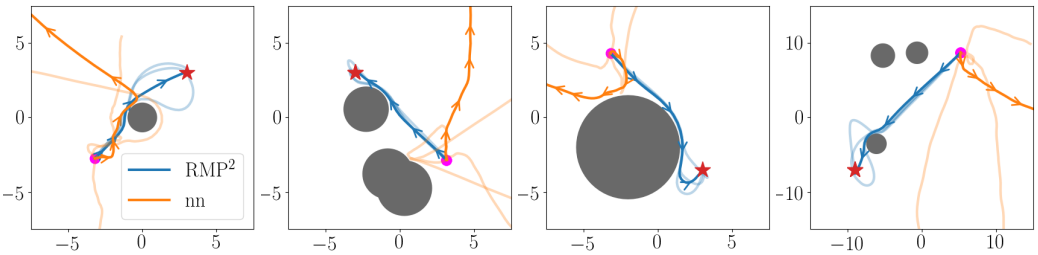


Figure 1: The RMPflow policy and neural-net policy performing a reaching task with a point robot, where each trajectory shows a policy with a random initialization of network weights. The policies are tested on environments within (left 2) and outside (right 2) the training distribution. The RMPflow policy succeeds consistently whereas the neural-net policy only learns to avoid collision.

References

[1] J. Achiam, D. Held, A. Tamar, and P. Abbeel. Constrained policy optimization. In *International Conference on Machine Learning*, pages 22–31, 2017.

[2] C. A. Cheng. *Efficient and principled robot learning: theory and algorithms*. PhD thesis, Georgia Institute of Technology, 2020.

[3] C.-A. Cheng, M. Mukadam, J. Issac, S. Birchfield, D. Fox, B. Boots, and N. Ratliff. RMPflow: A computational graph for automatic motion policy generation. *Proceedings of the 13th Annual Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2018.

[4] C.-A. Cheng, X. Yan, N. Ratliff, and B. Boots. Predictor-corrector policy optimization. In *International Conference on Machine Learning*, pages 1151–1161, 2019.

[5] J. Choi, F. Castañeda, C. J. Tomlin, and K. Sreenath. Reinforcement learning for safety-critical control under model uncertainty, using control lyapunov functions and control barrier functions. *arXiv preprint arXiv:2004.07584*, 2020.

[6] Y. Chow, O. Nachum, A. Faust, E. Duenez-Guzman, and M. Ghavamzadeh. Lyapunov-based safe policy optimization for continuous control. *arXiv preprint arXiv:1901.10031*, 2019.

[7] B. Christianson. Automatic hessians by reverse accumulation. *IMA Journal of Numerical Analysis*, 12(2):135–150, 1992.

[8] G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru, and Y. Tassa. Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757*, 2018.

[9] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[10] S. M. Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002.

[11] D. Kappler, F. Meier, J. Issac, J. Mainprice, C. Garcia Cifuentes, M. Wüthrich, V. Berenz, S. Schaal, N. Ratliff, and J. Bohg. Real-time perception meets reactive motion generation. *IEEE Robotics and Automation Letters*, 3(3):1864–1871, 2018. URL `https://arxiv.org/abs/1703.03512`.

[12] A. Li, C.-A. Cheng, B. Boots, and M. Egerstedt. Stable, concurrent controller composition for multi-objective robotic tasks. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 1144–1151. IEEE, 2019.

[13] A. Li, M. Mukadam, M. Egerstedt, and B. Boots. Multi-objective policy generation for multi-robot systems using Riemannian motion policies. In *International Symposium on Robotics Research*, 2019.

[14] X. Meng, N. Ratliff, Y. Xiang, and D. Fox. Neural autonomous navigation with riemannian motion policy. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8860–8866. IEEE, 2019.

[15] M. Mukadam, C.-A. Cheng, D. Fox, B. Boots, and N. Ratliff. Riemannian motion policy fusion through learnable lyapunov function reshaping. In *Conference on Robot Learning*, pages 204–219, 2019.

[16] C. Paxton, N. Ratliff, C. Eppner, and D. Fox. Representing robot task plans as robust logical-dynamical systems. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[17] M. A. Rana, A. Li, H. Ravichandar, M. Mukadam, S. Chernova, D. Fox, B. Boots, and N. Ratliff. Learning reactive motion policies in multiple task spaces from human demonstrations. In *Conference on Robot Learning*, 2019.

[18] N. D. Ratliff, J. Issac, D. Kappler, S. Birchfield, and D. Fox. Riemannian motion policies. *arXiv preprint arXiv:1801.02854*, 2018.

[19] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[20] A. Stooke, J. Achiam, and P. Abbeel. Responsive safety in reinforcement learning by pid lagrangian methods. *arXiv preprint arXiv:2007.03964*, 2020.

[21] G. Sutanto, N. Ratliff, B. Sundaralingam, Y. Chebotar, Z. Su, A. Handa, and D. Fox. Learning latent space dynamics for tactile servoing. In *2019 International Conference on Robotics and Automation (ICRA)*, 2019.

[22] C. Tessler, D. J. Mankowitz, and S. Mannor. Reward constrained policy optimization. In *International Conference on Learning Representations*, 2018.

[23] B. Wingo, C. Cheng, M. Murtaza, M. Zafar, and S. Hutchinson. Extending riemmanian motion policies to a class of underactuated wheeled-inverted-pendulum robots. In *IEEE International Conference on Robotics and Automation*, 2020.

A RMPflow

Here we describe the message passing steps of RMPflow [3]. The algorithm is based on two components: *1*) the RMP-tree: a directed tree encoding the structure of the task map; and *2*) the RMP-algebra: a set of operations to propagate information across the RMP-tree. In the RMP-tree, a node $\mathtt{u}$ stores the state $(\mathbf{x}_\mathtt{u}, \dot{\mathbf{x}}_\mathtt{u})$ on a manifold $\mathcal{M}_\mathtt{u}$ and the associated RMP $(\mathbf{a}_\mathtt{u}, \mathbf{M}_\mathtt{u})^{\mathcal{M}_\mathtt{u}}$, and an edge $\mathtt{e}$ represents to a smooth map $\psi_\mathtt{e}$ from the manifold of a parent node manifold to its child node manifold. The root node of the RMP-tree (denoted as $\mathtt{r}$) and the leaf nodes correspond to the configuration space $\mathcal{C}$ and the subtask spaces $\{\mathcal{T}_k\}$ on which the user-defined RMPs are hosted, respectively. The RMP-algebra comprises of three operators, which are for propagating information on the RMP-tree. For illustration, we consider the node $\mathtt{u}$ on manifold $\mathcal{M}$ with coordinates $\mathbf{x}$ and its M child nodes ($\mathtt{v}_m$ on manifold $\mathcal{N}_m$ with coordinates $\mathbf{y}_m$ for $m = 1, \ldots, M$) in the RMP-tree.

- (*i*) `pushforward` propagates the state of a node $(\mathbf{x}, \dot{\mathbf{x}})$ in the RMP-tree to update the states of its M child nodes $\{\mathbf{y}_m, \dot{\mathbf{y}}_m\}_{m=1}^M$. The state of its mth child node is computed as $(\mathbf{y}_m, \dot{\mathbf{y}}_m) = (\psi_m(\mathbf{x}), \mathbf{J}_m(\mathbf{x})\,\dot{\mathbf{x}})$, where ψ_m is the smooth map of the edge connecting the two nodes and $\mathbf{J}_m = \partial_\mathbf{x}\psi_m$ is the Jacobian matrix.
- (*ii*) `pullback` propagates $\{[\mathbf{f}_{\mathtt{v}_m}, \mathbf{M}_{\mathtt{v}_m}]^{\mathcal{N}_m}\}_{m=1}^M$, the RMPs of the child nodes in the natural form, to the parent node as $[\mathbf{f}_\mathtt{u}, \mathbf{M}_\mathtt{u}]^\mathcal{M}$:

$$\mathbf{f}_\mathtt{u} = \textstyle\sum_{m=1}^M \mathbf{J}_{\mathtt{e}_m}^\top (\mathbf{f}_{\mathtt{v}_m} - \mathbf{M}_{\mathtt{v}_m}\dot{\mathbf{J}}_{\mathtt{e}_m}\dot{\mathbf{x}}), \qquad \mathbf{M}_\mathtt{u} = \textstyle\sum_{m=1}^M \mathbf{J}_{\mathtt{e}_m}^\top \mathbf{M}_{\mathtt{v}_m}\mathbf{J}_{\mathtt{e}_m}. \tag{4}$$

  The natural form of RMPs are used here since they more efficient to combine.
- (*iii*) `resolve` maps an RMP from its natural form $[\mathbf{f}_\mathtt{u}, \mathbf{M}_\mathtt{u}]^\mathcal{M}$ to its canonical form $(\mathbf{a}_\mathtt{u}, \mathbf{M}_\mathtt{u})^\mathcal{M}$ by $\mathbf{a}_\mathtt{u} = \mathbf{M}_\mathtt{u}^\dagger \mathbf{f}_\mathtt{u}$, where $\dagger$ denotes Moore-Penrose inverse.

RMPflow (in Algorithm 2) computes the policy $\pi(\mathbf{q}(t), \dot{\mathbf{q}}(t)) = \mathbf{a}_\mathtt{r}$ on the configuration space $\mathcal{C}$ through the following procedure. Given the state $(\mathbf{q}(t), \dot{\mathbf{q}}(t))$ of the configuration space $\mathcal{C}$ at time t, the `pushforward` operator is first recursively applied to the RMP-tree to propagate the states up to the leaf nodes. Then, the subtask RMPs are evaluated on the leaf nodes and combined recursively backward along the RMP-tree by the `pullback` operator. The `resolve` operator is finally applied on the root node $\mathtt{r}$ to compute the desired acceleration $\mathbf{a}_\mathtt{r}$.

Algorithm 2 RMPflow Algorithm (Message Passing) [3]

Input: root state $(\mathbf{q}, \dot{\mathbf{q}})$, RMP-tree, `rmp_eval`
Return: motion policy $\pi(\mathbf{q}, \dot{\mathbf{q}})$
$nodes \leftarrow$ RMP-tree.$topologically\_sorted\_nodes$
// forward pass
For $node$ **in** $nodes$: *// from root to leaves*
    For $child$ **in** $node.children$:
        $child.state \leftarrow$ `pushforward`$(node.state)$
// evaluate leaf RMPs
For $node$ **in** RMP-tree.$leaves$:
    $node.rmp \leftarrow$ `rmp_eval(`$node.state$`)`
// backward pass
For $node$ **in** `reversed`$(nodes)$: *// from leaves to root*
    $node.rmp \leftarrow$ `pullback` $(node.child\_rmps)$
// resolve for the motion policy
$\pi(\mathbf{q}, \dot{\mathbf{q}}) \leftarrow$ `resolve`(RMP-tree.$root.rmp$)

B The Reverse Accumulation Algorithm for Jacobian Vector Product

The reverse accumulation algorithm to compute Jacobian-vector-product in RMP^2 is listed in Algorithm 3. The algorithm computes Jacobian-vector product through 2 passes. An auxiliary all ones vector $\boldsymbol{\lambda}$ is created and the gradient with respect to $\boldsymbol{\lambda}$ is tracked (line 2). In the first pass, the scalar $\boldsymbol{\lambda}_i^\top \mathbf{y}$ is computed, and then the auxiliary value $\mathbf{z} = (\partial_\mathbf{x}(\boldsymbol{\lambda}^\top \mathbf{y}))^\top$ is obtained through backward pass (line 4). Note that, numerically,

$$z_j = \partial_{x_j}\left(\textstyle\sum_k \lambda_k\, y_k\right) = \textstyle\sum_k \lambda_k \left(\partial_{x_j} y_k\right)$$

where z_j denotes the jth element of $\mathbf{z}$, same applies to other vectors, e.g. $\mathbf{x}$, $\mathbf{y}$ and $\boldsymbol{\lambda}$.

In the second pass (line 5), the Jacobian-vector product is computed as $(\partial_{\mathbf{x}}\mathbf{y})\mathbf{v} = (\partial_{\boldsymbol{\lambda}}\mathbf{z}^\top\mathbf{v})^\top$, since

$$\partial_{\lambda_i}\mathbf{z}^\top\mathbf{v} = \partial_{\lambda_i}\left(\sum_j z_j v_j\right) = \partial_{\lambda_i}\left(\sum_j\sum_k \lambda_k\left(\partial_{x_j}y_k\right)v_j\right) = \sum_j\left(\partial_{x_j}y_i\right)v_j = \left(\partial_{\mathbf{x}}y_i\right)\mathbf{v}.$$

Algorithm 3 `jvp(y,x,v)`

1: **Input:** $\mathbf{y}$, $\mathbf{x}$, $\mathbf{v}$
2: **Return:** $(\partial_{\mathbf{x}}\mathbf{y})\,\mathbf{v}$
3: $\boldsymbol{\lambda} \leftarrow \mathbf{1}$ *// create dummy variable for reverse accumulation*
4: $\mathbf{z} \leftarrow$ `gradient(`$\boldsymbol{\lambda}^\top\mathbf{y}$`,x)` *// compute sum of partial derivatives*
5: compute Jacobian-vector product $(\partial_{\mathbf{x}}\mathbf{y})\,\mathbf{v} \leftarrow$ `gradient(`$\mathbf{z}^\top\mathbf{v}$`,`$\boldsymbol{\lambda}$`)`

C Complexity

Consider a graph-structured task map with N nodes, where each node has dimension in $O(d)$ and has at most b parents. We suppose that $L \leq N$ nodes are leaf nodes, and that the automatic differentiation library is based on reverse-mode automatic differentiation.

We first analyze the complexity of task map evaluation and Jacobian-vector-product subroutine (Algorithm 3) based on reverse accumulation in preparation for the complexity analysis for RMP^2.

task map evaluation: For each node in the graph, the input and output dimensions are bounded by $O(bd)$ and $O(d)$, respectively. Hence, evaluating each node has a time complexity in $O(bd^2)$. Because each node is evaluated exactly once in in computing the full task map, the total time complexity of task map evaluation $O(Nbd^2)$. If the Gradient Oracle will be called (as in RMP^2 and the Direct algorithm), the value of each node needs to be stored in preparation for the gradient computation. Overall this would require a space complexity in $O(Nd)$ to store the values in the entire graph.

Jacobian-vector-product with L output nodes: Suppose that the output of the graph in Algorithm 3, $\mathbf{y}$, is a collection of L nodes in the graph. During reverse accumulation, the task map is first computed, which, based on the previous analysis, has time and space complexity of $O(Nbd^2)$ and $O(Nd)$, respectively. The dummy variable $\boldsymbol{\lambda}$ is of size $O(Ld)$ and computing the inner product $\boldsymbol{\lambda}^\top\mathbf{y}$ requires $O(Ld)$ computation (i.e. it creats a new node of dimension 1 with $2L$ parent nodes of dimension in $O(d)$). By the reverse-mode automatic differentiation assumption, the first backward pass on the graph (line 4) has time complexity of $O(Nbd^2 + Ld) = O(Nbd^2)$ and space complexity of $O(Nd + Ld) = O(Nd)$ [9]. The final backward pass (line 5) is on a graph of size $O(N)$, as the first backward pass creates additional $O(N)$ nodes. With similar analysis, the second backward pass have time complexity of $O(Nbd^2 + Ld) = O(Nbd^2)$ and space complexity of $O(Nd + Ld) = O(Nd)$. Therefore, the time and space complexity of Algorithm 3 is $O(Nbd^2)$ and $O(Nd)$, respectively.

C.1 RMP^2

Forward pass: The complexity of line 3–5 in Algorithm 1 follow the precursor analyses above. Here the computation graph is always of size $O(N)$ (the original task map is in $O(N)$ and each call of Gradient Oracle in the Jacobian-vector-product subroutine creates additional $O(N)$ nodes in the computation graph). By previous analysis, the time and space complexity of the forward pass are $O(Nbd^2)$ and $O(Nd)$, respectively.

Leaf evaluation: Assume, for each leaf node, $O(d^3)$ computation is needed for evaluating the metric and $O(d^2)$ for acceleration. The leaf evaluation step then have time complexity of $O(Ld^3)$ and space comlexity of $O(Ld^2 + Ld) = O(Ld^2)$.

Backward pass: By previous analysis, task map evaluation requires $O(Nbd^2)$ computation and $O(Nd)$ space. The vector-matrix-vector product for computing $\mathbf{v}$ and $\mathbf{w}$ has time complexity of $O(Ld^2)$. To compute the metric at root, $\mathbf{M}_\mathtt{r}$, the first backward pass, `gradient(v,q)` needs $O(Nbd^2)$ time and $O(Nd)$ space as it operates on a graph of size $O(N)$ where the number of parents of each node is in $O(b)$[a]. The `jacobian` operator in line 10 is done by $O(d)$ sequential calls of

[a]Except the final node aggregating L outputs. However, it does not change the complexity as it adds a complexity in $O(Ld^2) < O(Nbd^2)$.

Algorithm 4 Direct Algorithm 2

Input: root state $(\mathbf{q}, \dot{\mathbf{q}})$, task map ψ, `rmp_eval`
Return: motion policy $\pi(\mathbf{q}, \dot{\mathbf{q}})$
// forward pass
$\{\mathbf{x}_i\}_{i=1}^L \leftarrow$ `task_map(`$\mathbf{q}$`)`
$\{\mathbf{J}_i\}_{i=1}^L \leftarrow$ `jacobian(`$\{\mathbf{x}_i\}_{i=1}^L$`,`$\mathbf{q}$`)`
$\{\dot{\mathbf{x}}_i\}_{i=1}^L \leftarrow$ `jvp(`$\{\mathbf{x}_i\}_{i=1}^L$`,`$\mathbf{q}$`,`$\dot{\mathbf{q}}$`)` *// leaf node velocity*
$\{\mathbf{c}_i\}_{i=1}^L \leftarrow$ `jvp(`$\{\dot{\mathbf{x}}_i\}_{i=1}^L$`,`$\mathbf{q}$`,`$\dot{\mathbf{q}}$`)` *// curvature terms*
// evaluate leaf RMPs
evaluate leaf RMPs $\{(\mathbf{M}_i, \mathbf{f}_i)\}_{i=1}^L \leftarrow$ `rmp_eval(`$\{(\mathbf{x}_i, \dot{\mathbf{x}}_i)\}_{i=1}^L$`)`
// backward pass
$\mathbf{M}_\mathrm{r} \leftarrow \sum_{i=1}^L \mathbf{J}_i^\top \mathbf{M}_i \mathbf{J}_i, \quad \mathbf{f}_\mathrm{r} \leftarrow \sum_{i=1}^L \mathbf{J}_i^\top (\mathbf{f}_i - \mathbf{M}_i \mathbf{c}_i)$ *// compute root RMP*
// resolve for the motion policy
$\pi(\mathbf{q}, \dot{\mathbf{q}}) \leftarrow \mathbf{M}_\mathrm{r}^\dagger \mathbf{f}_\mathrm{r}$

the Gradient Oracle. Hence, it has time complexity of $O(Nbd^3)$ and space complexity of $O(Nd)$. (Because we are not taking further derivatives, the values of the new graphs created in calling the `jacobian` operator do not need to be stored.) With similar analysis, computing $\mathbf{f}_\mathrm{r}$ requires $O(Nbd^3)$ computation and $O(Nd)$ space.

Resolve: The matrix inversion has time complexity of $O(d^3)$ and space complexity of $O(d^2)$.

In summary, the time complexity of RMP^2 is $O(Nbd^2 + Ld^3 + Nbd^3 + d^3) = O(Nbd^3)$ and the space complexity is $O(Nd + Ld^2)$.

C.2 RMPflow

First we need to convert a graph with $O(b)$ parent nodes into a tree. This can be done by creating meta nodes that merge all the parents of a node into a single parent node; inside the mega node, each component is computed independently. Therefore, for a fair comparison, in the following analysis, we shall assume that in the tree version evaluating each node would need a time complexity in $O(bd^2)$. We suppose the space complexity to store all the nodes is still in $O(Nd)$ because the duplicated information resulting from the creation of the meta nodes can be handled by sharing the same memory reference in a proper implementation [3].

Forward pass: During the forward pass, similar to the reverse accumulation analysis, $O(bd^2)$ per node is needed for `pushforward`, i.e. computing the pushforward velocity through reverse accumulation, yielding a time complexity of $O(Nbd^2)$. The space complexity is $O(Nd)$ for storing the state at every node.

Leaf evaluation: Same as RMP^2.

Backward pass: Computing the metric in (4) requires $O(bd^3)$ computation per node, yielding time complexity of $O(Nbd^3)$ and space complexity of $O(Nd^2)$. The curvature term $\dot{\mathbf{J}}_{\mathtt{e}_m}\dot{\mathbf{x}}$ can be computed through reverse accumulation similar to RMP^2, which has time complexity of $O(bd^2)$ per node. The matrix-vector products to compute the force requires $O(bd^2)$ space and computation for each node.

Resolve: Same as RMP^2.

Thus, the time complexity of RMPflow is $O(Nbd^2 + Ld^3 + Nbd^3 + Nbd^2 + d^3) = O(Nbd^3)$ and the space complexity is $O(Nd + Ld^2 + Nd^2 + bd^2 + d^2) = O(Nd^2 + Ld^2)$.

D Experiment Details

D.1 Environment Setup

The RL example considers a point robot on a 2-d plane. The *state* of the robot is composed of the coordinate on the plane, $\mathbf{x} \in \mathbb{R}^2$, as well as the velocity, $\dot{\mathbf{x}} \in \mathbb{R}^2$. The robot is modeled as a discrete-time double integrator system,

$$\begin{aligned} \mathbf{x}_{t+1} &= \mathbf{x}_t + \dot{\mathbf{x}}_t \cdot dt, \\ \dot{\mathbf{x}}_{t+1} &= \dot{\mathbf{x}}_t + \mathbf{u}_t \cdot dt, \end{aligned} \tag{5}$$

where $\mathbf{u} \in \mathbb{R}^2$ is the control input, and $dt = 0.1$ is the sampling time for the discrete-time system.

The environment has a randomly generated goal $\mathbf{g} \in \mathbb{R}^2$ and up to 3 randomly generated circular obstacles $\{\mathcal{O}_i\}_{i=1}^K$. An obstacle is denoted as $\mathcal{O}_i \coloneqq (\mathbf{o}_i, r_i)$, where $\mathbf{o}_i \in \mathbb{R}^2$ is the center of the obstacle $\mathcal{O}_i$ and r_i is the radius. At every step t, the observation given to the robot consists of $\{\mathbf{x}_t, \dot{\mathbf{x}}_t, \mathbf{g}, \{\mathcal{O}_i\}_{i=1}^K\}$, and the robot applies action $\mathbf{u}_t = \pi(\mathbf{x}_t, \dot{\mathbf{x}}_t, \mathbf{g}, \{\mathcal{O}_i\}_{i=1}^K)$. The reward given to the robot at each step is the negative Euclidean distance to the goal, with an additional penalty of -100 if the robot collides with an obstacle. Collision checking operates in a finer time-scale of $dt/10$. The robot is considered in collision with an obstacle $\mathcal{O}_i$ if and only if $\|x - \mathbf{o}_i\| \leq r_i$. In the training phase, due to efficiency considerations, collision checking is only done in one randomly sampled frame out of the 10 frames of each step, and a collision penalty of 1000 is given to the robot if the robot is in collision with any obstacle at the sampled frame. This sampled reward is the same as the true reward in expectation. For the sake of simplicity, the dynamics of the system (5) does not change even if the robot is in collision with the obstacle. However, the large negative reward of collision should discourage the robot to collide with any obstacles.

Upon environment reset, the initial state of the robot, the goal, and the obstacles are randomly sampled. During training, these quantities are sampled from the following distributions: $\mathbf{x}_0 \sim \text{uniform}(-0.1, 0.1)^2$, $\dot{\mathbf{x}}_0 \sim \text{uniform}(-0.05, 0.05)^2$, $\mathbf{g} \sim \text{uniform}(-5, 5)^2$, $K \sim \text{uniform}(\{0, 1, 2, 3\})$, and $\mathbf{o}_i \sim \text{uniform}(-5, 5)^2$, $r_i \sim \text{uniform}(1, 2)$ for $i = 1, \ldots, K$.

D.1.1 The RMPflow Policy

Goal reaching: The goal-reaching policy is parameterized as a metric neural network and a acceleration neural network. The input to the networks is the collection of state and goal position, i.e. $\{\mathbf{x}, \dot{\mathbf{x}}, \mathbf{g}\}$. Both networks are 2-layer neural-nets with 32 hidden units and `elu` activation functions. The output layer of the metric network uses `tanh` activation function to ensure that the metric is always bounded, which further guarantees collision-free behavior when combined with the collision avoidance policy below. To produce a positive semi-definite metric, the output of the metric network is reshaped to a square matrix $\mathbf{A} \in \mathbb{R}^{2\times 2}$ and the output metric is computed as $\mathbf{M}_g = \mathbf{A}\mathbf{A}^\top$.

Collision avoidance: We use the barrier-type collision avoidance policy [3, Section 5.1] to provide safety guarantee of the robot. The policy operates on the distance space between the robot and each obstacle, $\mathbf{z}_i = d(\mathbf{x}, \mathcal{O}_i) = \|\mathbf{x} - \mathbf{o}_i\|_2 - r_i$. The collision avoidance policy [3] is given by,

$$\mathbf{M}_{o_i} = \frac{0.2 + 2\min(0, \dot{\mathbf{z}}_i)\dot{\mathbf{z}}_i}{\mathbf{z}^4}, \quad \mathbf{a}_{o_i} = \frac{4 \times 10^{-5}}{\mathbf{z}_i^2(0.2 + 2\min(0, \dot{\mathbf{z}}_i)\dot{\mathbf{z}}_i)} + 2\frac{(0.2 + \min(0, \dot{\mathbf{z}}_i)\dot{\mathbf{z}}_i)\dot{\mathbf{z}}_i^2}{(0.2 + 2\min(0, \dot{\mathbf{z}}_i)\dot{\mathbf{z}}_i)\mathbf{z}_i}.$$

In practice, the metric and acceleration outputs are clipped at a large value to avoid numerical issues such overflow and ill-conditioned metrics.

Damping: To ensure the numerical stability of the `resolve` step, a damping policy is provided so that the metric at the root node is always positive-definite. The damping policy is given by $\mathbf{M}_d(\mathbf{x}, \dot{\mathbf{x}}) \equiv 0.1I$, and $\mathbf{a}_d(\mathbf{x}, \dot{\mathbf{x}}) = -\dot{\mathbf{x}}$.

D.1.2 The Neural-net Policy

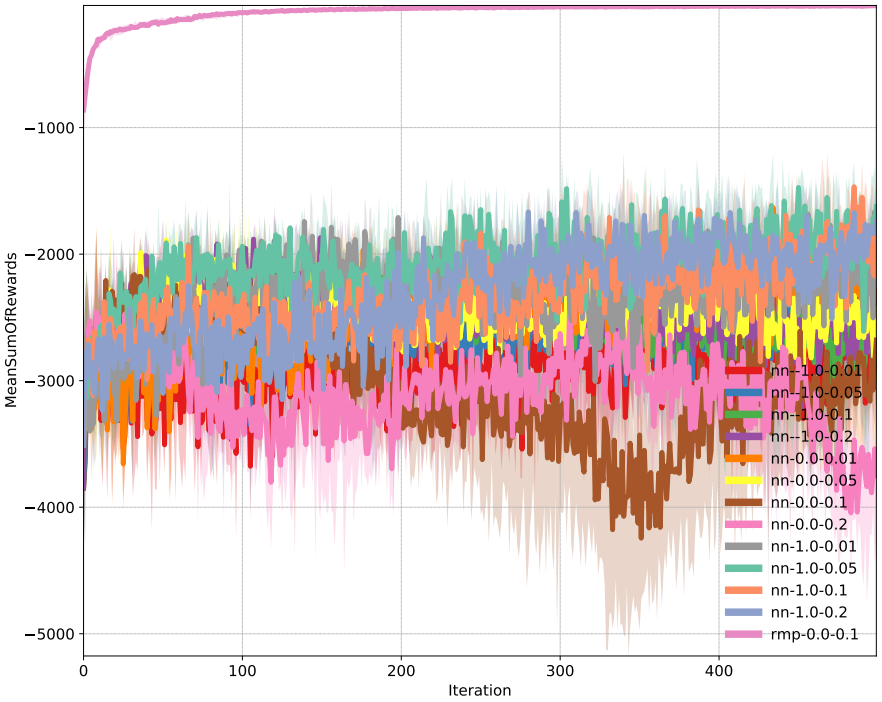
The neural-net policy directly parameterize $\mathbf{u}_t = \pi_\theta(\mathbf{x}_t, \dot{\mathbf{x}}_t, \mathbf{g}, \{\mathcal{O}_i\}_{i=1}^K)$. We use a feed-forward neural network with 2 hidden layers of size 64 with `tanh` activation functions. To ensure that the observation dimension is kept fixed among environments with different number of obstacles, place holder obstacles with radius -1 are provided.

D.2 Learning Setup

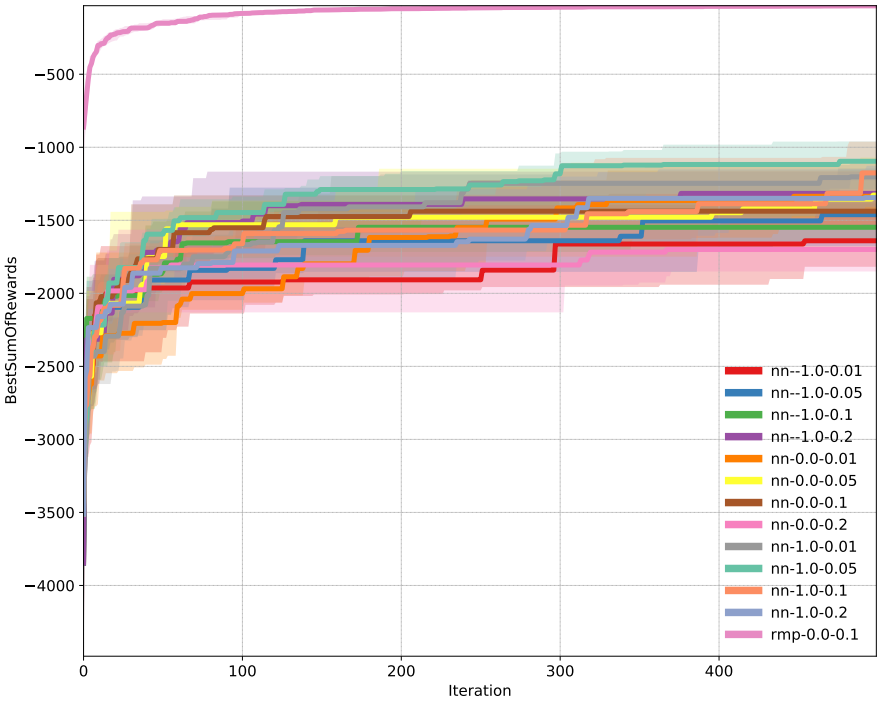
Algorithm: We adopt the adaptive implementation of the natural policy gradient in [4, Appendix E]. For each deterministic policy described above, we construct a Gaussian policy by setting the deterministic policy as the mean of the Gaussian and setting the covariance to have a diagonal structure. Both the mean and the covaraince are optimized during the training. In each iteration, the Gaussian policy is rolled out in the environment to collect trajectory data and then the GAE policy gradient estimate [19] is computed, where the value function estimate is a fully connected neural network with two hidden layers (of size 256 and 128, respectively) and `tanh` activation functions and it is trained on-policy using Monte-Carlo estimates of the cumulative rewards after each policy

update. This GAE gradient estimate is then used as the first-order direction for the natural policy gradient update.

Hyperparameters: We tested a variety of hyperparameters for the neural-net policy: natural policy gradient of step size ranging in $\{0.01, 0.05, 0.1, 0.2\}$ and initial standard deviation for the Gaussian policy in $\{e^{-1}, 1, e\}$. The cumulative reward for the policy at each training iteration and the best policy upon each iteration for each of the hyperparameter are shown in Figure 2. We chose the best performing set of hyperparameters, with step size of 0.05 and initial standard deviation of 2.72 (`nn-1.0-0.05`), for the experiments in the main text. The RMPflow policy (`rmp-0.0-0.1`) achieves significant higher reward throughout learning compared to the neural-net policy. (We tried training the RMPflow policy using the default hyperparameter in the codes of [4], which resulted into the training curve shown in the figure. As it significantly outperforms the neural-net policies, we did not search additional hyperparameters for the RMPflow policy to further improve its performance).



(a) Sum of reward of the policy at each iteration



(b) Sum of reward of the best policy upon each iteration

Figure 2: The cumulative reward of (a) the policy at each training iteration and (b) the best policy in each iteration. The curve shows the median over 8 random seeds while the shaded area is given by the first and third quantiles. The legend shows the method (RMPflow or neural-net policy), the initial log standard deviation for the Gaussian policies, and the learning rate for natural gradient descent. The neural-net policy with initial standard deviation of 2.72, and learning of 0.05 (`nn-1.0-0.05` in the figures) is chosen for the experiments in the main text. The RMPflow policy (`rmp-0.0-0.1`) achieves significant higher reward throughout learning compared to the neural-net policy.