DEEP SEQUENCED LINEAR DYNAMICAL SYSTEMS FOR MANIPULATION POLICY LEARNING

Mohammad Nomaan Qureshi
IIIT Hyderabad
mohammad.nomaan@research.iiit.ac.in

Ben Eisner & David Held
Carnegie Mellon University
{baeisner,dheld}@andrew.cmu.edu

ABSTRACT

In policy learning for robotic manipulation tasks, action parameterization can have a major impact on the final performance and sample efficiency of a policy. Unlike highly-dynamic continuous-control tasks, many manipulation tasks can be efficiently performed by a sequence of simple, smooth end-effector motions. Building on this intuition, we present a new class of policies built on top of differentiable Linear Dynamical System (dLDS) units, our differentiable formulation of the classical LDS. Constructing policies using dLDS units yields several advantageous properties, including trajectory coherence across timesteps, stability, and invariance under translation and scaling. Inspired by the sequenced LDS approach proposed by Dixon & Khosla (2004), we propose a deep neural-network policy parameterization based on sequenced dLDS units, and we integrate this policy class into standard on-policy reinforcement learning settings. We conduct extensive experiments on Metaworld environments and show a notable improvement in performance and sample efficiency compared to other state-of-the-art algorithms. Additional visualizations and code can be found at `https://sites.google.com/view/deep-sequenced-lds`.

1 INTRODUCTION

Imagine a robot trying to pour water into a glass from a bottle. First, it has to reach the bottle's cap, open the cap, move the bottle towards the glass, and then pour the water into the glass. The complex trajectory that the robot follows to accomplish this task can be decomposed into a sequence of simpler trajectories. Even when we change the positions of glass and bottle, the underlying simpler trajectories follow approximately the same shape, but are transformed accordingly. In this paper, we explore whether we can parameterize a learned policy that is biased towards producing smooth trajectories with a similar decomposable, transformable structure.

In order to achieve smooth, coherent trajectories when the underlying action space does not implicitly provide them, a policy must jointly reason about an agent's actions across time steps. One popular approach to achieve this is to predict sequences of actions together at a lower frequency than the actions are executed - for instance, by directly repeating a single action ("frame skip" Braylan et al. (2015)), or jointly predicting low-level actions for the next k timesteps Bahl et al. (2020). However, repeating the same action for k steps achieves smoothness without local expressiveness; and jointly predicting k low-level actions achieves local expressiveness but produces actions that are not necessarily coherent or smooth.

Recently, Bahl et al. (2020) proposed a policy representation (NDP) that is guaranteed to produce smooth, coherent trajectories by reparametrizing the action space with parameters of a non-linear dynamical system based on Dynamic Motion Primitives (DMPs), which consist of a set of forcing function weights and a goal state Ijspeert et al. (2013). They then use a standard trajectory-following controller to follow the trajectory for a fixed number of time steps before re-evaluating the policy. While the underlying Dynamic Motion Primitives used in NDPs can express very complex trajectories, their performance is sensitive to the choice of basis function and number of basis functions, and they aren't inherently invariant to translation or rotation of the state space Saveriano et al. (2021). With these limitations in mind, outputting DMP parameters to provide a wide variety of trajectories for different environmental variations – particularly in goal-conditioned manipulation tasks – presents a challenging optimization problem.

To address these limitations, we propose a different action parameterization based on a Linear Dynamical System (LDS). This choice of dynamical system achieves guaranteed scale and translational invariance (proof in Appendix B), structurally improving the generalization of goal-conditioned policies. Based on the intuition that many manipulation tasks may be accomplished by a *sequence* of simple skills, we decompose an episode into a sequence of LDS units, in which we learn a neural-network based policy to predict the goal for each unit. This action space parameterization is heavily inspired by the classical sequential LDS formulation proposed by Dixon & Khosla (2004), who observed that a complex trajectory can be decomposed into a sequence of simpler linear dynamical systems. However, their policy formulation is not differentiable, making it non-trivial to adapt to a deep reinforcement learning setting. Therefore, we replace the LDS unit with a differentiable version (dLDS), and we propose a corresponding sequenced dLDS policy learning algorithm that can be trained using any policy-gradient algorithm. We conduct thorough experiments on Metaworld Yu et al. (2019) environments and show a notable improvement in sample efficiency in comparison to other state-of-the-art algorithms. To summarize our contributions:

- We introduce a differentiable formulation of training Linear Dynamical Systems which can be integrated into any gradient-based learning algorithm.
- We propose a novel RL policy composed from a sequence of dLDS units.
- We conduct thorough experiments in numerous Metaworld environments and show a notable improvement in sample efficiency compared to baseline policy-learning algorithms.

2 Related Work

Dynamical systems for manipulation Dynamic motion primitives have been classically used for solving complex manipulation tasks Ijspeert et al. (2013); Tamosiunaite et al. (2011); Ijspeert et al. (2013); Thota et al. (2016). More recently, methods have employed DMPs with Neural Networks in order to learn policies that could output the parameters of the underlying dynamical system (DMP), which is then use to sample trajectories Bahl et al. (2020); Pahič et al. (2018; 2020); Bahl et al. (2021). Hierarchial NDPs Bahl et al. (2021) builds upon NDPs with a hierarchial framework which can work on unseen task configurations.

Reinforcement learning for manipulation tasks Along with recent progress in deep reinforcement learning, there has been increased interest in applying reinforcement learning to manipulation tasks. Several popular benchmark tasks widely used in RL literature feature challenging goal-conditioned manipulation tasks such as Fetch and MetaWorld, Andrychowicz et al. (2017); Yu et al. (2019). Several methods have leveraged fully on-robot training to accomplish generalized manipulation tasks using robot arms equipped with parallel-jaw grippers Gu et al. (2017); Kalashnikov et al. (2018). RL has also been used for dynamic manipulation tasks Zeng et al. (2020) and manipulation tasks involving high-dimensional control OpenAI et al. (2018; 2019). Several methods have also sought to encode explicit inductive biases for manipulation tasks directly into the policy structure, including graph neural networks Li et al. (2020), motion planners Yamada et al. (2020), and through the use of action primitives Dalal et al. (2021) .

Residual Reinforcement Learning: Recent papers Johannink et al. (2018); Silver et al. (2018) proposed combining a hand-engineered control policy with a reinforcement learning policy. This combination can solve tricky tasks that can be solved approximately by a hand-engineered policy but can only be solved completely using the help of a residual RL policy applied on top. However, coming up with such an underlying hand-engineered policy for each task is complex and labor-intensive. Our sequenced linear dynamical system policy can also be viewed as coming up with a trainable hard-coded structure accompanied by a residual reinforcement learning policy. Residual RL has been explored for manipulation tasks Johannink et al. (2018); Silver et al. (2018); Zeng et al. (2020); Garcia-Hernando et al. (2020); Ranjbar et al. (2021); Schoettler et al. (2019) and in a Learning from Demonstration setting Alakuijala et al. (2021); Schoettler et al. (2019).

3 Preliminaries

Reinforcement Learning: The aim of applying reinforcement learning (RL) is to learn to generate actions that maximize the cumulative reward on a given task. More specifically, RL aims to find optimal parameters θ for the policy π_θ which can maximise the average discounted return J:

$$\mathbf{J}^{\pi} = E_{a_t \sim \pi_{\theta(s_t)}, s_0 \sim p_0}[\textstyle\sum_{t=1}^{t=T} \gamma^t r(s_t, a_t)] \tag{1}$$

where s_t represents current state, a_t represents action sample from policy π_θ, r(s_t, a_t) represents the reward function and T is the maximum episode length. Policy Gradient algorithms approximate the gradient of J with respect to policy parameters θ as

$$\nabla_\theta \mathbf{J} = E_{a_t \sim \pi_{\theta(s_t)}}[(G_t - V(s_{t+1}))\nabla \log \pi_\theta(a_t|s_t)] \tag{2}$$

where $V(s)$ represents value function and $G_t = r_{t+1} + \gamma V(s_{t+1})$. Most modern algorithms parametrize the policy π_θ as a neural network, and hence θ represents the network parameters Schulman et al. (2017); Mnih et al. (2016).

Linear Dynamical Systems: Dixon & Khosla (2004) present a way to represent trajectories using Linear Dynamical Systems. Consider a trajectory $\mathbf{Y} = [y_1, y_2, \ldots, y_T]$ where each state y_t is an m-dimensional vector and $T \geq m$. Dixon & Khosla (2004) provide a least squares solution for extracting these LDS representations from a given demonstration. An LDS can be represented using the difference equation:

$$\mathbf{y}_{t+1} = \mathbf{R} \cdot (\mathbf{y_t} - \mathbf{y_T}) + \mathbf{y_t} \tag{3}$$

where $\mathbf{R}$ is a matrix that parameterizes the LDS. For a complete trajectory Y, We can rewrite Eq. 3 in a stacked matrix equation as:

$$[\mathbf{y}_2, ..., \mathbf{y}_T] = \mathbf{R} \cdot ([\mathbf{y_1}, ..., \mathbf{y_{T-1}}] - [\mathbf{y_T}, ..., \mathbf{y_T}]) + [\mathbf{y}_1, ..., \mathbf{y}_{T-1}] \tag{4}$$

or equivalently as

$$\mathbf{Y}[2:T] = \mathbf{R} \cdot (\mathbf{Y[1:T-1]} - \mathbf{\Gamma_T}) + \mathbf{Y}[1:T-1] \tag{5}$$

where $\Gamma_T = [y_T, ..., y_T]$. The least-squares solution for is:

$$\hat{\mathbf{R}} = (\mathbf{Y}[2:T] - \mathbf{Y}[1:T-1]) \cdot (\mathbf{Y}[1:T-1] - \Gamma_T)^P \tag{6}$$

where, for any matrix M, $M^P = M^T \cdot (M \cdot M^T)^{-1}$. Here y_T represents the goal state or "attractor" of the trajectory. The estimated LDS matrix $\hat{\mathbf{R}}$ captures important properties of the provided trajectory $\mathbf{Y}$. A simple trajectory can be represented using a single LDS. A complex trajectory can be represented using a sequence of LDS's.

The LDS induces a control law which maps the current state to the desired state of the robot. Consider a robot with end-effector position y_t and end effector velocity $\dot{y}_t$. An LDS matrix $\mathbf{R}$ induces a control field which maps each current end-effector position y_t to a desired velocity $\dot{y}_t$. Given a goal state g^*, an LDS matrix $\mathbf{R}$ can be used to predict $\dot{y}_t$.

$$\mathbf{y}_{t+1} = \mathbf{R} \cdot (\mathbf{y_t} - \mathbf{g}^*) + \mathbf{y_t} \tag{7}$$

$$\dot{\mathbf{y}}_t = \mathbf{y_{t+1}} - \mathbf{y_t} \tag{8}$$

4 METHOD

4.1 DIFFERENTIABLE LDS

For the LDS system in equations 7 and 8 to converge towards state g^*, the eigenvalues of $\mathbf{R}$ have to lie in the range $(-1, 0)$. We utilize this property to reparametrize the $\mathbf{R}$ matrix. Consider Q, an n-dimensional full rank matrix, and Λ, an n-dimensional diagonal matrix. We can use the eigen-structure to construct the LDS matrix $\mathbf{R}$ from its eigenvalues (Λ) and eigenvectors (Q) as :

$$\Lambda_s := -sigmoid(\Lambda) \tag{9}$$

$$\mathbf{R} := Q \cdot \Lambda_s \cdot Q^{-1} \tag{10}$$

Here Λ and Q are the learnable parameters used to construct an LDS matrix $\mathbf{R}$. Defined in this way, the eigenvalues of $\mathbf{R}$, given by Λ_s, are guaranteed to be in the range $(-1, 0)$ as required. This re-parametrization of the LDS matrix in terms of its eigenvalues and eigenvectors can help us integrate it into any end-to-end gradient-based method.

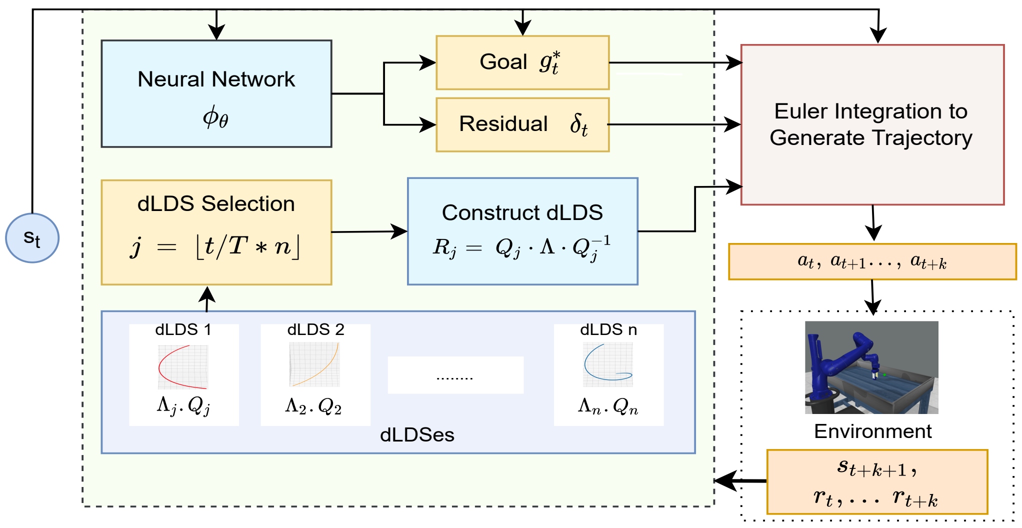

Figure 1: An overview of an RL policy based on differentiable Linear Dynamic System (dLDS) units. Every k timesteps, the neural network ϕ_θ estimates a goal g_t and a residual δ_t, which are used to parameterize one of the n dLDS units. This dLDS unit is then integrated for k steps using Euler integration to generate a sequence of k actions, which are applied in sequence to the environment.

4.2 dLDS Extraction from a Reference Trajectory

In this subsection, we demonstrate how we can extract dLDS representations in a learning from demonstration (LfD) setting using a batched gradient descent algorithm. Given a reference trajectory Y = $[y_1, y_2, \ldots\ y_T]$, we randomly initialize the dLDS parameters Q, Λ. Given the current position y_t, we can get the next state estimate $\hat{y}_t$ as

$$\hat{\mathbf{g}}^* = \mathbf{y}_T \tag{11}$$

$$\hat{\mathbf{y}}_{t+1} = \mathbf{R} \cdot (\mathbf{y_t} - \mathbf{g}^*) + \mathbf{y_t} \tag{12}$$

Here, the goal g^* is the same as y_T, the trajectory's final position. We can compute the loss as the mean squared error between predicted next state $\hat{y}_{t+1}$ and the actual next state y_{t+1}.

$$\textbf{loss} = \frac{1}{(T-1)} * \sum_{t=1}^{t=T-1} ||(\mathbf{R} \cdot (\mathbf{y_t} - \mathbf{g}^*) + \mathbf{y_t} - \mathbf{y_{t+1}})|| \tag{13}$$

We can calculate the gradients for parameters Q, Λ according to the above-defined loss function. Algorithm 1 in the Appendix summarizes our dLDS learning formulation when a reference trajectory is available.

4.3 Deep Sequenced LDS Policies

In this section we explain how dLDS can be integrated with a standard on-policy reinforcement learning algorithm like PPO Schulman et al. (2017). We decompose an episode with length T, into a sequence of n dLDS units which can be represented by $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_n$. dLDS parameters (Λ_i, Q_i) can be used to construct $\mathbf{R}_i$ using Equations 9 and 10. Our policy also contains a goal estimator ϕ_θ, which takes state s_t as input and predicts the goal g_t^* and residual δ_t. In this work, we select a deep neural network as the model class for ϕ_θ, although any estimator suitable for policy gradient methods is compatible with our method.

$$g_t^*, \delta_t = \phi(s_t) \tag{14}$$

Once we get the goal g_t^*, we need an dLDS matrix R_j to generate a trajectory from the current end effector position y_t to g_t. We use the current time-step within the episode to choose a dLDS from n

dLDS's; specifically, at time-step t, dLDS index j is calculated as

$$j = \lfloor t/T \cdot n \rfloor \tag{15}$$
$$\Lambda_s = -sigmoid(\Lambda_j) \tag{16}$$
$$\mathbf{R}_j = Q_j \cdot \Lambda_s \cdot Q_j^{-1} \tag{17}$$

where T is maximum episode length.

Now using goal g_t^*, end effector position y_t, and LDS matrix R_j, we can perform Euler Integration to estimate the next state:

$$\dot{y}_t = \mathbf{R}_j \cdot (y_t - g_t^*) + \delta_t \tag{18}$$
$$y_{t+1} = y_t + \dot{y}_t * \Delta_t \tag{19}$$
$$\ddot{y}_t = \dot{y}_t - \dot{y}_{t-1} \tag{20}$$

We perform this integration for k discrete time steps and use the trajectory obtained to calculate the actions from our policy. We assume that our action can be represented as a target velocity $\dot{y}$ or position $\dot{y}\Delta t$.

A key difference between dLDS policies and NDPs Bahl et al. (2020) is that in NDPs, a neural network predicts $\phi_{ndp}(s_t)$ outputs the parameters w_t of an underlying dynamical system (DMP) at each inference step:

$$g_t^{ndp}, w_t = \phi_{ndp}(s_t) \tag{21}$$

In contrast, our estimation module $\phi_{lds}(s_t)$ only outputs some of the LDS parameters, namely the goal and residual (Eq 14). The LDS matrices ($\Lambda_{1..n}$ and $Q_{1..n}$) of our sequenced dLDS system (Eqs 15, 16, 17) are higher-level parameters that are a function of the "segment" that the trajectory is in; they do not need to be predicted at each timestep. These parameters are still updated during training using gradient descent:

$$\Lambda_{1..n} = \Lambda_{1..n} + \alpha \cdot \nabla J(\Lambda_{1..n}) \tag{22}$$
$$Q_{1..n} = Q_{1..n} + \alpha \cdot \nabla J(Q_{1..n}) \tag{23}$$

This presents our policy with a simpler optimization problem of only learning the mapping at each timestep from state to goals and residuals rather than outputting the entire non-linear dynamical system at each inference-step. To further verify this design choice, we compare our method against a policy that outputs the LDS parameters at each timestep from the deep network, rather than learning them at training time and then keeping them fixed per task.

4.4 Goal Generation for dLDS

As described in Equation 14, our learned policy generates the goals for the dLDS at each timestep. Rather than outputting just a single goal, we found it more effective to have the network output n goals at each timestep ($g_1, g_2, ..., g_n$), where n is the number of dLDS's. We then select the goal g_j according to the index of the current LDS (Equation 15). We found out that this works much better than outputting just a single goal at each timestep. This is expected since when the network predicts a single goal, it has to figure out which dLDS it is predicting the goal for. However, when we predict n goals, the policy has to solve a more straightforward problem of predicting goal states for each dLDS (without needing to reason about exactly which dlDS the robot is in at the moment).

4.5 Residual Prediction

Apart from predicting goal states for each LDS, we also predict a residual δ_t for each step in the policy rollout. We take inspiration from Johannink et al. (2018), where the idea is to combine a hand-engineered control structure with a residual RL policy. Similarly, we treat our underlying dLDS structure as the basis on top of which a residual can be learned. However, parameters of our underlying dLDS structure are learnable, unlike that of Johannink et al. (2018) where one has to come up with a hand-engineered policy for each task involved. We empirically verify that combining the action output $\dot{y}_t$ of dLDS with a residual δ_t gives a decent boost in its performance.

4.6 Training details

To train our dLDS policy, we generate k actions to apply to the environment for each of the next k timesteps. We use PPO Schulman et al. (2017) to optimise the policy parameters (neural network parameters and dLDS parameters). However, any other policy gradient algorithm could be used. Algorithm 2 in the Appendix summarises our approach. Hyperparameters and other details can be found in the appendix.

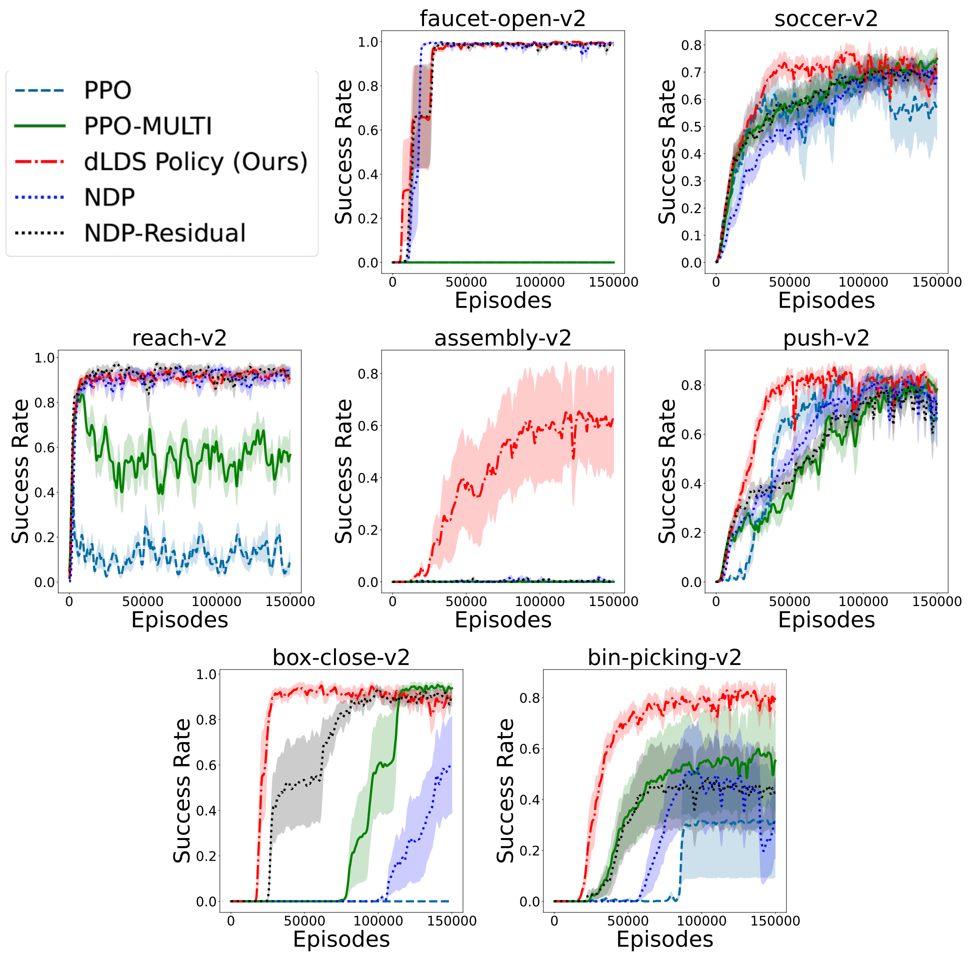

Figure 2: **Sample Efficiency:** Our algorithm is significantly more sample-efficient than baselines on several environments, and performs at least as well as the baselines on every task. Here, the y-axis is the success rate, while the x-axis represents the number of environment episodes. We compare our algorithm against PPO Schulman et al. (2017), Multi-action PPO, NDP Bahl et al. (2020), and NDP with residual. We use 3 dLDS's for our dLDS policy in each of these tasks.

5 Results

The motivation behind our work is to reparameterize a reinforcement learning policy with a decomposable, transformable structure so as to simplify the underlying policy learning problem. We conduct an extensive experimental evaluation of our method on seven Metaworld-v2 Yu et al. (2019) environments, namely 'faucet-open-v2', 'reach-v2', 'soccer-v2', 'assembly-v2', 'push-v2', 'box-close-v2' and 'bin-picking-v2'. Our policy's action space is the same as that of Metaworld-v2 environments, which consists of end-effector position change and "grab-effort" which is the normalized torque the fingers should apply.

We compare the performance of our **dLDS Policy** with other State of the art on-policy algorithms:

- **PPO (raw)**: Vanilla Proximal Policy Optimisation Schulman et al. (2017) without any modifications.
- **PPO-Multi**: A multi-action version of PPO, which outputs actions for the next 'k' timesteps.
- **NDP**: Neural Dynamic Policy Bahl et al. (2020) which builds on Dynamic Motion Primitives.
- **NDP-Residual**: A modified version of NDP with a learned residual.

5.1 Policy Learning Results

Figure 2 shows the comparison of our dLDS Policy with the various other baselines. We obtained a significant improvement in sample efficiency compared to the other baseline algorithms. For the tasks 'faucet-open-v2' and 'reach-v2', both NDP and dLDS Policy have similar performance, while for other tasks, dLDS policy outperforms other algorithms. dLDS Policy outperforms the other baselines consistently across all tasks. PPO is competitive for task 'push-v2' and 'soccer-v2'. However, its performance does not scale to other tasks. Multi-PPO is comparatively better than PPO but fails to learn the tasks 'faucet-open-v2' and 'assembly-v2'. NDP is competitive in 'reach-v2' and 'faucet-open-v2'. However, its performance in harder tasks ('assembly-v2', 'box-close-v2' and 'bin-picking-v2') is significantly lower than that of the dLDS Policy. The Residual version of NDP (NDP-Residual) shows comparative performance to NDP and dLDS in tasks 'faucet-open-v2' and 'reach-v2'. NDP-Residual) outperforms the vanilla-NDP algorithm in other tasks except 'assembly-v2', where both the algorithms fail to learn anything meaningful. We provide our algorithm with a decomposable, transformable structure which results in an easy optmisation problem.

5.2 Ablations

We now study the effect of various design choices on the performance our policy.

5.2.1 Differentiable LDS vs. Dynamically predicted LDS

Similar to NDP, we train a version of our policy in which the deep part of policy outputs the parameters of the underlying dynamical system (in our case, parameters Q and Λ) at each inference-step instead of learning them globally for each task. We call this ablation "O-LDS policy." This is an inherently harder optimization problem since the network has to determine the goal and shape the trajectory concurrently. We can verify our intuition from Figure 3 that learning the dLDS parameters (Q, Λ) globally for each task instead of outputting them at each timestep from a deep network simplifies the learning problem.

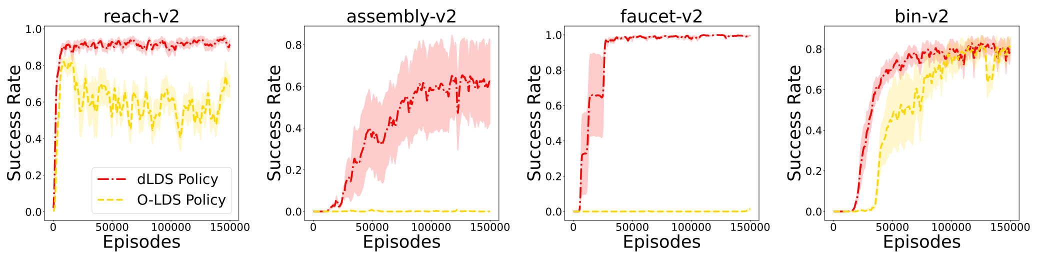

Figure 3: What happens when we output the LDS parameters every timestep using a deep network (O-LDS policy), instead of making them global parameters that are learned per task (dLDS Policy)? The above figure compares the performance of the two versions. We can see that dLDS policy is easily able to outperform the O-LDS policy

5.2.2 Number of dLDS

The number of dLDS's n is a key hyperparameter of dLDS policies. Figure 4 compares the performance of dLDS Policy as we vary the number of dLDS's. The number of LDS's controls the expressiveness of the policy. For example, when n = 1, the policy is constrained to express very

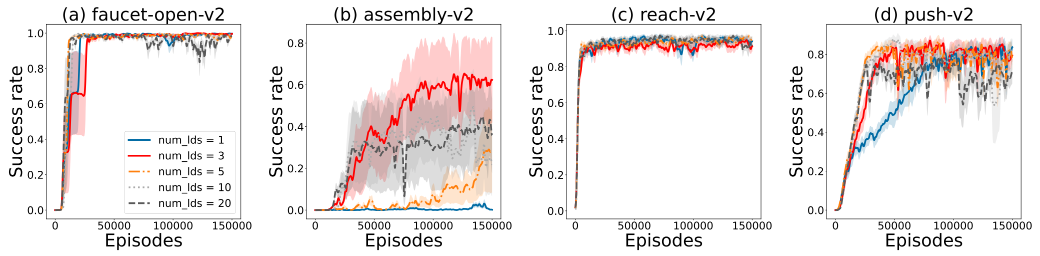

Figure 4: We compare the effect of varying the number n of dLDS's available to the policy on policy performance. We find that across our evaluation tasks, $n = 3$ consistently achieves the highest performance. This provides evidence that chaining together a small number of simple motion primitives is an effective way to complete this class of manipulation tasks. For simpler tasks like 'faucet-open-v2' and 'reach' even a single dLDS based policy was simple enough to achieve high performance!

simple trajectories. This dLDS (n=1) policy might be sufficient for a simpler task like 'reach-v2' and 'faucet-open-v2' but cannot learn to perform a complex task like 'bin-picking-v2' or 'assembly-v2', where policy needs to figure out a sequence of meaningful skills to accomplish the task. Similarly, having a very high number of dLDS's (e.g., 20) makes the policy too complex, and hence its performance is affected (Figure 4(b)). We find out that n=3 is sufficient for most metaworld tasks.

5.2.3 Residual vs. No Residual

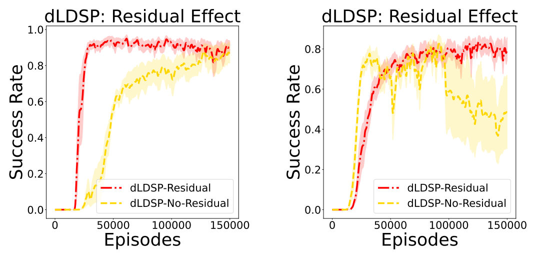

Figure 5: Effect of Predicting a Residual Term: **Left** is 'box-close-v2'. **right**: 'bin-picking-v2'. We can see that including a residual term significantly improves the sample efficiency and performance of the dLDS policy.

How does adding a residual term affect the performance of a dynamical-system-based policy? We find that adding a residual term to the output of a dynamical system-based policy – for both dLDS and or NDP – can improve performance substantially on a range of tasks. Figure 5 compares the effect of residual on the performance of our dLDS Policy. We observe that incorporating a residual improves sample-efficiency and performance on several environments, and on others is as least as good as the dLDS policy without a residual. This result also holds when we train a residual version of NDPs, showing similar improvements over NDP on the 'box-close-v2'. and 'bin-picking-v2' tasks.

6 Conclusion

Our paper introduces a novel reinforcement learning policy with a decomposible and transformable structure, which can generate smooth robot trajectories. At the heart of our approach lies the novel differentiable LDS (dLDS) formulation. We re-parametrize the LDS in terms of eigen-properties, which helps us to learn the LDS parameters directly in an end-to-end fashion. We then propose a novel dLDS based reinforcement learning policy, which integrates these dLDS's with a standard on-policy algorithm (PPO). We perform extensive experiments and show a notable improvement in sample efficiency of our algorithm compared to other policy parameterizations. Last, we perform various ablations to verify several design choices of our algorithm.

Choosing simpler models (i.e. LDS units) for parts of tasks where complexity is lower is a first step towards encoding meaningful inductive biases into manipulation policies in reinforcement learning settings. We hope that our use of dynamical systems for policy representation inspires further investigation into this promising direction. Specifically, we hope to investigate a more flexible tradeoff between learning a fixed skill bank of dynamical policies and predicting a new dynamical system at each time-step. Additionally, we hope to explore how explicitly sequencing dynamical systems may improve policy transfer, generalization, and learning from demonstrations.

REFERENCES

Minttu Alakuijala, Gabriel Dulac-Arnold, Julien Mairal, Jean Ponce, and Cordelia Schmid. Residual reinforcement learning from demonstrations, 2021.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Openai Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In I Guyon, U V Luxburg, S Bengio, H Wallach, R Fergus, S Vishwanathan, and R Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Shikhar Bahl, Mustafa Mukadam, Abhinav Gupta, and Deepak Pathak. Neural dynamic policies for end-to-end sensorimotor learning. In *NeurIPS*, 2020.

Shikhar Bahl, Abhinav Gupta, and Deepak Pathak. Hierarchical neural dynamic policies, 2021.

Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Frame skip is a powerful parameter for learning to play atari. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

Murtaza Dalal, Deepak Pathak, and Russ R Salakhutdinov. Accelerating robotic reinforcement learning via parameterized action primitives. *Adv. Neural Inf. Process. Syst.*, 34, December 2021.

K.R. Dixon and P.K. Khosla. Trajectory representation using sequenced linear dynamical systems. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 4, pp. 3925–3930 Vol.4, 2004. doi: 10.1109/ROBOT.2004.1308881.

The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. `https://github.com/rlworkgroup/garage`, 2019.

Guillermo Garcia-Hernando, Edward Johns, and Tae-Kyun Kim. Physics-based dexterous manipulations with estimated hand poses and residual reinforcement learning, 2020.

Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3389–3396, May 2017.

Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. Dynamical movement primitives: Learning attractor models for motor behaviors. *Neural Computation*, 25(2): 328–373, 2013. doi: 10.1162/NECO_a_00393.

Tobias Johannink, Shikhar Bahl, Ashvin Nair, Jianlan Luo, Avinash Kumar, Matthias Loskyll, Juan Aparicio Ojea, Eugen Solowjow, and Sergey Levine. Residual reinforcement learning for robot control. *CoRR*, abs/1812.03201, 2018. URL `http://arxiv.org/abs/1812.03201`.

Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. QT-Opt: Scalable deep reinforcement learning for Vision-Based robotic manipulation. June 2018.

Richard Li, Allan Jabri, Trevor Darrell, and Pulkit Agrawal. Towards practical Multi-Object manipulation using relational reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4051–4058, May 2020.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous In-Hand manipulation. August 2018.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand. October 2019.

Rok Pahič, Andrej Gams, Aleš Ude, and Jun Morimoto. Deep encoder-decoder networks for mapping raw images to dynamic movement primitives. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5863–5868, 2018. doi: 10.1109/ICRA.2018.8460954.

Rok Pahič, Barry Ridge, Andrej Gams, Jun Morimoto, and Aleš Ude. Training of deep neural networks for the generation of dynamic movement primitives. *Neural Networks*, 127:121–131, 2020. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2020.04.010. URL `https://www.sciencedirect.com/science/article/pii/S0893608020301301`.

Alireza Ranjbar, Ngo Anh Vien, Hanna Ziesche, Joschka Boedecker, and Gerhard Neumann. Residual feedback learning for contact-rich manipulation tasks with uncertainty, 2021.

Matteo Saveriano, Fares J Abu-Dakka, Aljaz Kramberger, and Luka Peternel. Dynamic movement primitives in robotics: A tutorial survey. February 2021.

Gerrit Schoettler, Ashvin Nair, Jianlan Luo, Shikhar Bahl, Juan Aparicio Ojea, Eugen Solowjow, and Sergey Levine. Deep reinforcement learning for industrial insertion tasks with visual inputs and natural rewards, 2019.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL `http://arxiv.org/abs/1707.06347`.

Tom Silver, Kelsey R. Allen, Josh Tenenbaum, and Leslie Pack Kaelbling. Residual policy learning. *CoRR*, abs/1812.06298, 2018. URL `http://arxiv.org/abs/1812.06298`.

Minija Tamosiunaite, Bojan Nemec, Aleš Ude, and Florentin Wörgötter. Learning to pour with a robot arm combining goal and shape learning for dynamic movement primitives. *Robotics and Autonomous Systems*, 59(11):910–922, 2011. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2011.07.004. URL `https://www.sciencedirect.com/science/article/pii/S0921889011001254`.

Pavan Kumar Thota, Harish chaandar Ravichandar, and Ashwin P. Dani. Learning and synchronization of movement primitives for bimanual manipulation tasks. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pp. 945–950, 2016. doi: 10.1109/CDC.2016.7798389.

Jun Yamada, Youngwoon Lee, Gautam Salhotra, Karl Pertsch, Max Pflueger, Gaurav S Sukhatme, Joseph J Lim, and Peter Englert. Motion planner augmented reinforcement learning for robot manipulation in obstructed environments. October 2020.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning (CoRL)*, 2019. URL `https://arxiv.org/abs/1910.10897`.

Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics, 2020.

A Algorithms

Algorithm 1 Differentiable Linear Dynamical System

```
Require: Reference Trajectory Y = $[y_1, y_2, ..., y_T]$, learning_rate $\alpha$
 1: Initialise $\Lambda, Q$                                    ▷ Initialise LDS Matrix parameters.
 2: while $\|y_t - g^*\| \geq \epsilon$ do                       ▷ Convergence criterion
 3:     $y_t :=$ get-current-obs()                             ▷ Obtain the current State information
 4:     $\Lambda_s = -sigmoid(\Lambda)$                        ▷ Normalize Eigen values to be negative and less than 1
 5:     $R := Q \cdot \Lambda_s \cdot Q^{-1}$                  ▷ Construct R using its Eigen vecturs and Eigen values
 6:     $\hat{Y} := R \cdot (g^* - Y[: -1]) + Y[: -1]$         ▷ Predict $y_{t+1}$ from $y_t$
 7:     loss = $MSE(\hat{Y}, Y[1 :])$                          ▷ Compute loss wrt reference trajectory.
 8:     loss.backward()                                         ▷ Compute Gradients
 9:     $Q = Q - \alpha \frac{dloss}{dQ}$                      ▷ Update parameters
10:     $\Lambda = \Lambda - \alpha \frac{dloss}{d\Lambda}$
```

Algorithm 2 Differentiable LDS for Policy learning

```
Require: Policy $\pi$, number_of_lds n, max_episode_len T
Require: rollout_length k, discount_factor $\gamma$
 1: for 1, 2 ... episodes do
 2:     for t = 0, k, ..., T do
 3:         Get goal $g_t^*, \delta_t$ using Eq. 14
 4:         Select dLDS $j$ using Eq. 15
 5:         Create dLDS $R_j$ from $\Lambda_j, Q_j$ (Eq. 16 and 17)
 6:         for i = 0, 1, ..., k do
 7:             $\dot{y}_{t+i} = R_j \cdot (y_{t+i} - g_t^*) + \delta_i$
 8:             $y_{t+i+1} = \dot{y}_{t+i}dt + y_{t+i}$
 9:         Set Action $a_{t,...,t+k} = \dot{y}_{t,...,t+k}$
10:         Apply $a_{t,..,t+k}$ on Environment
11:         Get $s_{t+1,..,t+k+1}$, $r_{t,..,t+k}$
12:         $R_t$ = $r_t + \gamma^1 r_{t+1} + ... + \gamma^k r_{t+k}$
13:         Store Transition $(s_t, a_{t,..,t+k}, s_{t+k+1}, R_t)$
14:     end for
15:     Compute Policy Gradients $\nabla_\theta J$
16:     Update Network and dLDS parameters
17: end for
```

B Translational Invariance Proof

We present a proof to prove translational variance in dLDS's. Let's assume our LDS matrix is represented by R_j. At any discrete time-step t, the position is y_t and goal position is g. The position at time $t + 1$ is given by equation

$$y_{t+1} = R_j \cdot (y_t - g) + y_t \tag{24}$$

Now assuming we shift the origin of our coordinate frame by ∇o. The right hand side of Eq. 24 becomes

$$R_j \cdot ((y_t - \nabla o - (g - \nabla o)) + (y_t - \nabla o) \tag{25}$$

$$R_j \cdot (y_t - g) + y_t - \nabla o \tag{26}$$

$$y_{t+1} - \nabla o \tag{27}$$

Hence translational invariance is proved. A proof of scale invariance follows in the same way.

C IMPLEMENTATION DETAILS

We modified the RL Garage repository from garage contributors (2019) (https://github.com/rlworkgroup/garage) to support multi-action policies. dLDS's paramters learn at a higher learning rate than the neural network paramters of the policy. We do not change rest of the parameters given in garage's examples. We use Metaworld environments to test the performance of our algorithms. We use 'faucet-open-v2', 'reach-v2', 'soccer-v2', 'assembly-v2', 'push-v2', 'box-close-v2' and 'bin-picking-v2' from the larger suite of environments available in Metaworld. These environments were chosen randomly. We use a maximum_episode_length of 100 for each of these environments.

Hyper-parameters Details	
Hypyerparameter	Value
Learning Rate	3 x 10^{-4}
Discount Factor	0.99
Use GAE	True
Entropy Coefficient	0
Normalized Observation	True
Maximum Gradient Norm	0.5
PPO Mini-Batches	32
PPO Epochs	10
Clip Parameters	0.1
Optimizer	Adam
Batch Size	5000
Max Episode Length	100
Learning Rate for LDS parameters	$1x10^{-3}$

D ADDITIONAL ABALATIONS

In this section we present several additional ablations we performed to analyze the impact of various design decision on our policy formulation. Descriptions can be found in the captions of Figures 6, 7, 8, and 9.

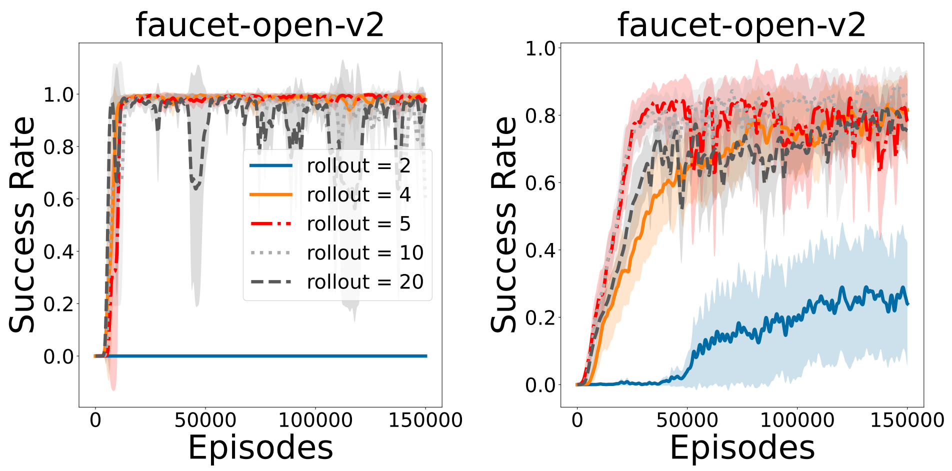

Figure 6: **Rollout Length Comparison**: Here we compare the effect of rollout length – the number of actions our LDS policy outputs at a time – on the performance of our policy. Similar to NDP, we find that a rollout length of 5 performs best across tasks.
.

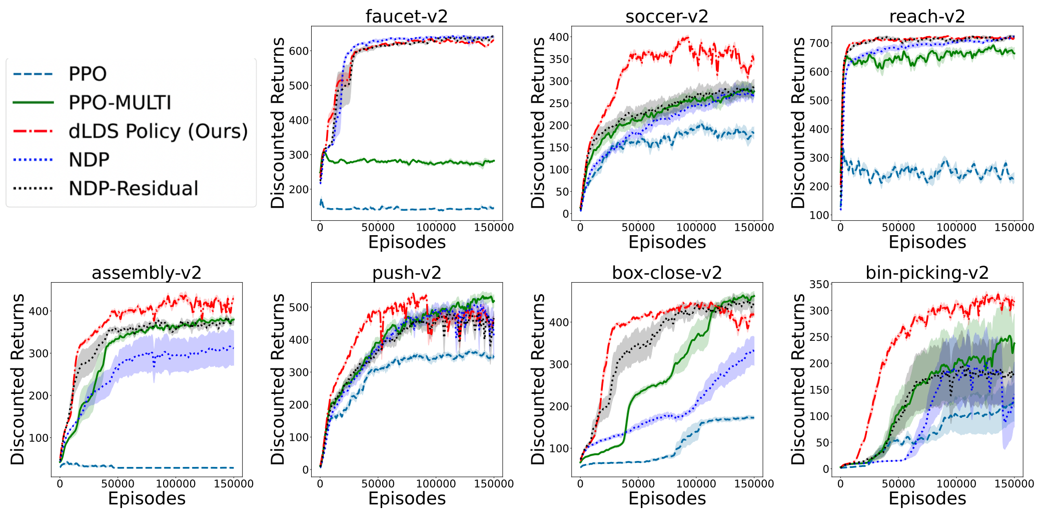

Figure 7: **Comparison of average discounted returns:** We compare average discounted return that different algorithms are able to achieve. These values are thresholded to compute the binary success values presented in the Figure 2. We can see that our novel dLDS policy is able to consistently outperforms baseline algorithms. Here the number of dLDS units $n = 3$ for each task.

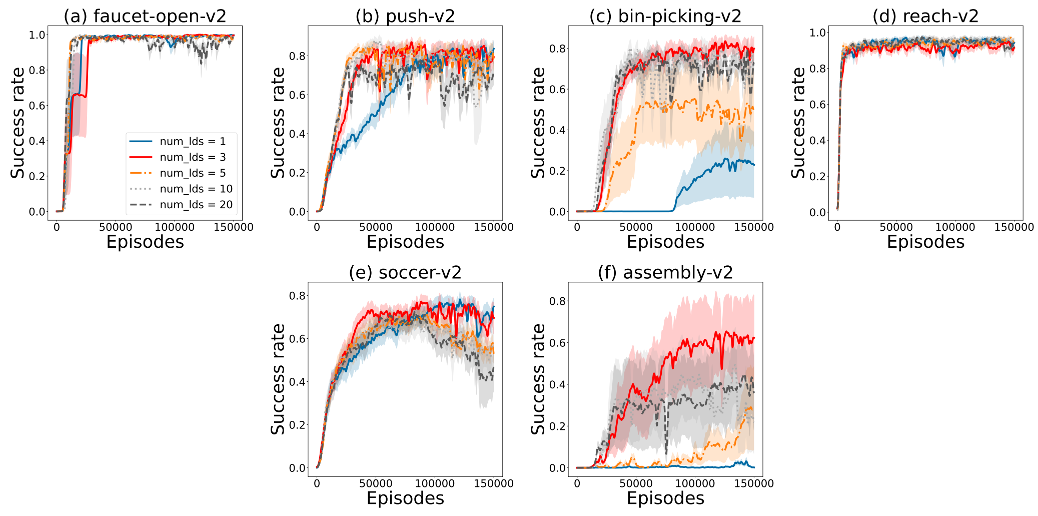

Figure 8: **Number of LDS's :** How does the performance vary with the number of dLDS's? This figure extends the results presented in Figure 4. For the task 'faucet-open-v2' and 'reach-v2' all the version of our policy (num˙lds=1, 3, 5, 10, 20) are competitive. This is expected, since the manipulation trajectory required to solve these tasks is very simple, and any number of primitives will solve this task quickly. However for complex tasks like 'assembly-v2' and 'bin-picking-v2', where the policy has to figure out a sequence of meaningful skills, access to a modest bank of dLDS's improves performance.

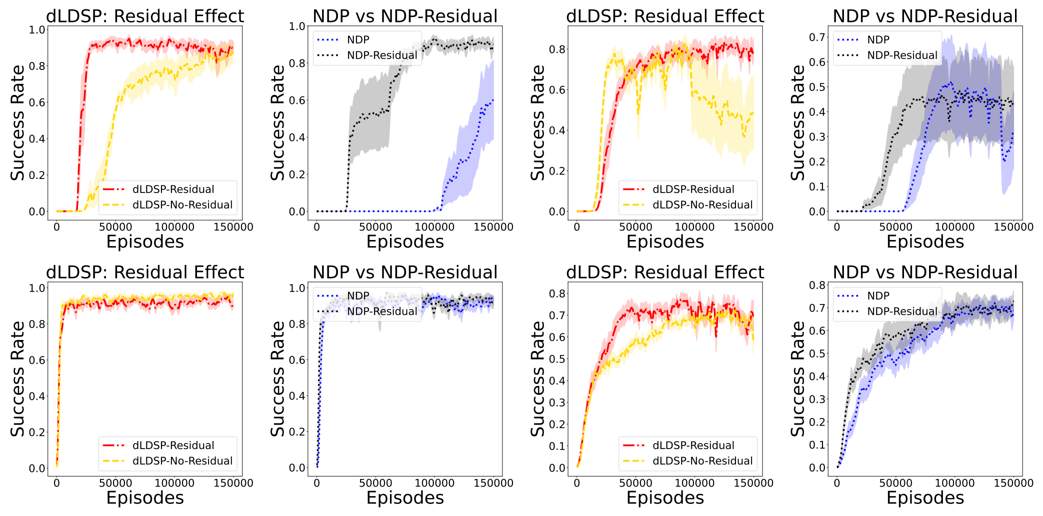

Figure 9: **Effect of Predicting a Residual Term**: Here we compare the effect of predicting a residual term on performance of dLDS Policy and NDPs. We observe that the residual term either improves the performance of dLDS and NDP policies or performs comparably.