# Neural Dynamic Policies for End-to-End Sensorimotor Learning

**Shikhar Bahl**
CMU

**Mustafa Mukadam**
FAIR

**Abhinav Gupta**
CMU

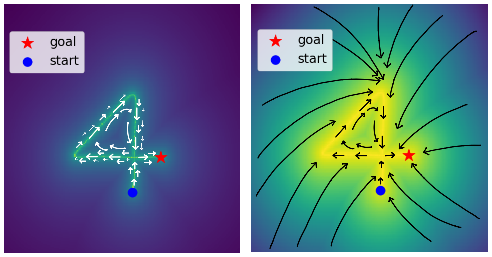
**Deepak Pathak**
CMU

## Abstract

The current dominant paradigm in sensorimotor control, whether imitation or reinforcement learning, is to train policies directly in raw action spaces such as torque, joint angle, or end-effector position. This forces the agent to make decisions individually at each timestep in training, and hence, limits the scalability to continuous, high-dimensional, and long-horizon tasks. In contrast, research in classical robotics has, for a long time, exploited dynamical systems as a policy representation to learn robot behaviors via demonstrations. These techniques, however, lack the flexibility and generalizability provided by deep learning or reinforcement learning and have remained under-explored in such settings. In this work, we begin to close this gap and embed the structure of a dynamical system into deep neural network-based policies by reparameterizing action spaces via second-order differential equations. We propose Neural Dynamic Policies (NDPs) that make predictions in trajectory distribution space as opposed to prior policy learning methods where actions represent the raw control space. The embedded structure allows end-to-end policy learning for both reinforcement and imitation learning setups. We show that NDPs outperform the prior state-of-the-art in terms of either efficiency or performance across several robotic control tasks for both imitation and reinforcement learning setups. Project video and code are available at: https://shikharbahl.github.io/neural-dynamic-policies/.

## 1 Introduction

Consider an embodied agent tasked with throwing a ball into a bin. Not only does the agent need to decide where and when to release the ball, but also needs to reason about the whole trajectory that it should take such that the ball is imparted with the correct momentum to reach the bin. This form of reasoning is necessary to perform many such everyday tasks. Common methods in deep learning for robotics tackle this problem either via imitation or reinforcement learning. However, in most cases, the agent's policy is trained in raw action spaces like torques, joint angles, or end-effector positions, which forces the agent to make decisions at each time step of the trajectory instead of making decisions in the trajectory space itself (see Figure 1). But then how do we reason about trajectories as actions?



Vanilla Policy

NDP (Ours)

Figure 1: Vector field induced by NDPs. The goal is to draw the planar digit 4 from the start position. The dynamical structure in NDP induces a smooth vector field in trajectory space. In contrast, a vanilla policy has to reason individually in different parts.

A good trajectory parameterization is one that is able to capture a large set of an agent's behaviors or motions while being physically plausible. In fact, a similar question is also faced by scientists

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

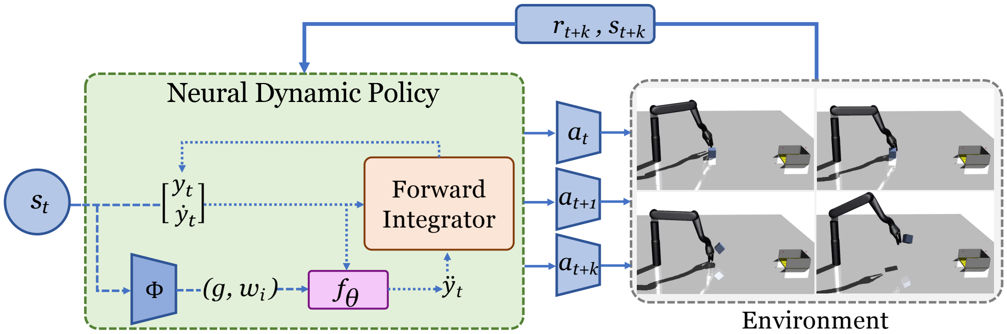


Figure 2: Given an observation from the environment, $s_t$, a Neural Dynamic Policy generates parameters $w$ (weights of basis functions) and $g$ (goal for the robot) for a forcing function $f_\theta$. An open loop controller then uses this function to output a set of actions for the robot to execute in the environment, collecting future states and rewards to train the policy.

while modeling physical phenomena in nature. Several systems in science, ranging from motion of planets to pendulums, are described by differential equations of the form $\ddot{y} = m^{-1} f(y, \dot{y})$, where $y$ is the generalized coordinate, $\dot{y}$ and $\ddot{y}$ are time derivatives, $m$ is mass, and $f$ is force. Can a similar parameterization be used to describe the behavior of a robotic agent? Indeed, classical robotics has leveraged this connection to represent task specific robot behaviors for many years. In particular, dynamic movement primitives (DMP) [20–22, 41] have been one of the more prominent approaches in this area. Despite their successes, DMPs have not been explored much beyond behavior cloning paradigms. This is partly because these methods tend to be sensitive to parameter tuning and aren't as flexible or generalizable as current end-to-end deep network based approaches.

In this work, we propose to bridge this gap by embedding the structure of dynamical systems[1] into deep neural network-based policies such that the agent can directly learn in the space of physically plausible trajectory distributions (see Figure 1(b)). **Our key insight is to reparameterize the action space in a deep policy network with nonlinear differential equations corresponding to a dynamical system and train it end-to-end over time in either reinforcement learning or imitation learning setups.** However, this is quite challenging to accomplish, since naively predicting a full arbitrary dynamical system directly from the input, trades one hard problem for another. Instead, we want to prescribe some structure such that the dynamical system itself manifests as a layer in the deep policy that is both, amenable to take arbitrary outputs of previous layers as inputs, and is also fully differentiable to allow for gradients to backpropagate.

We address these challenges through our approach, Neural Dynamic Policies (NDPs). Specifically, NDPs allow embedding desired dynamical structure as a layer in deep networks. The parameters of the dynamical system are then predicted as outputs of the preceding layers in the architecture conditioned on the input. **The 'deep' part of the policy then only needs to reason in the lower-dimensional space of building a dynamical system that then lets the overall policy easily reason in the space of trajectories.** In this paper, we employ the aforementioned DMPs as the structure for the dynamical system and show its differentiability, although they only serve as a design choice and can possibly be swapped for a different differentiable dynamical structure, such as RMPs [39].

We evaluate NDPs in imitation as well as reinforcement learning setups. NDPs can utilize high-dimensional inputs via demonstrations and learn from weak supervisory signals as well as rewards. In both settings, NDPs exhibit better or comparable performance to state-of-the-art approaches.

## 2 Modeling Trajectories with Dynamical Systems

Consider a robotic arm exhibiting a certain behavior to accomplish some task. Given a choice of coordinate system, such as either joint-angles or end-effector position, let the state of the robot be $y$, velocity $\dot{y}$ and acceleration $\ddot{y}$. In mechanics, Euler-Lagrange equations are used to derive the equations of motion as a general second order dynamical system that perfectly captures this behavior [43, Chapter 6]. It is common in classical robotics to represent movement behaviors with

[1] Dynamical systems here should not be confused with dynamics model of the agent. We incorporate dynamical differential equations to represent robot's behavioral trajectory and not physical transition dynamics.

such a dynamical system. Specifically, we follow the second order differential equation structure imposed by Dynamic Movement Primitives [22, 31, 41]. Given a desired goal state $g$, the behavior is represented as:

$$\ddot{y} = \alpha(\beta(g - y) - \dot{y}) + f(x), \tag{1}$$

where $\alpha, \beta$ are global parameters that allow critical damping of the system and smooth convergence to the goal state. $f$ is a non-linear forcing function which captures the shape of trajectory and operates over $x$ which serves to replace time dependency across trajectories, giving us the ability to model time invariant tasks, e.g., rhythmic motions. $x$ evolves through the first-order linear system:

$$\dot{x} = -a_x x \tag{2}$$

The specifics of $f$ are usually design choices. We use a sum of weighted Gaussian radial basis functions [22] shown below:

$$f(x, g) = \frac{\sum \psi_i w_i}{\sum \psi_i} x(g - y_0), \quad \psi_i = e^{(-h_i(x-c_i)^2)} \tag{3}$$

where $i$ indexes over $n$ which is the number of basis functions. Coefficients $c_i = e^{\frac{-i\alpha_x}{n}}$ are the horizontal shifts of each basis function, and $h_i = \frac{n}{c_i}$ are the width of each of each basis function. The weights on each of the basis functions $w_i$ parameterize the forcing function $f$. This set of nonlinear differential equations induces a smooth trajectory distribution that acts as an attractor towards a desired goal (see Figure 1, right). We now discuss how to combine this dynamical structure with deep neural network based policies in an end-to-end differentiable manner.

# 3 Neural Dynamic Policies (NDPs)

We condense actions into a space of trajectories, parameterized by a dynamical system, while keeping all the advantages of a deep learning based setup. We present a type of policy network, called Neural Dynamic Policies (NDPs) that given an input, image or state, can produce parameters for an embedded dynamical structure, which reasons in trajectory space but outputs raw actions to be executed. Let the unstructured input to robot be $s$, (an image or any other sensory input), and the action executed by the robot be $a$. We describe how we can incorporate a dynamical system as a differentiable layer in the policy network, and how NDPs can be utilized to learn complex agent behaviors in both imitation and reinforcement learning settings.

## 3.1 Neural Network Layer Parameterized by a Dynamical System

Throughout this paper, we employ the dynamical system described by the second order DMP equation (1). There are two key parameters that define what behavior will be described by the dynamical system presented in Section 2: basis function weights $w = \{w_1, \ldots, w_i, \ldots, w_n\}$ and goal $g$. NDPs employ a neural network $\Phi$ which takes an unstructured input $s$[2] and predicts the parameters $w, g$ of the dynamical system. These predicted $w, g$ are then used to solve the second order differential equation (1) to obtain system states $\{y, \dot{y}, \ddot{y}\}$. Depending on the difference between the choice of robot's coordinate system for $y$ and desired action $a$, we may need an inverse controller $\Omega(.)$ to convert $y$ to $a$, i.e., $a = \Omega(y, \dot{y}, \ddot{y})$. For instance, if $y$ is in joint angle space and $a$ is a torque, then $\Omega(.)$ is the robot's inverse dynamics controller, and if $y$ and $a$ both are in joint angle space then $\Omega(.)$ is the identity function.

As summarized in Figure 2, neural dynamic policies are defined as $\pi(a|s;\theta) \triangleq \Omega\big(\mathtt{DE}\big(\Phi(s;\theta)\big)\big)$ where $\mathtt{DE}(w, g) \rightarrow \{y, \dot{y}, \ddot{y}\}$ denotes solution of the differential equation (1). The forward pass of $\pi(a|s)$ involves solving the dynamical system and backpropagation requires it to be differentiable. We now show how we differentiate through the dynamical system to train the parameters $\theta$ of NDPs.

## 3.2 Training NDPs by Differentiating through the Dynamical System

To train NDPs, estimated policy gradients must flow from $a$, through the parameters of the dynamical system $w$ and $g$, to the network $\Phi(s;\theta)$. At any time $t$, given the previous state of robot $y_{t-1}$ and

[2] robot's state $y$ is not to be confused with environment observation $s$ which contains world as well as robot state (and often velocity). $s$ could be given by either an image or true state of the environment.

velocity $\dot{y}_{t-1}$ the output of the DMP in Equation (1) is given by the acceleration

$$\ddot{y}_t = \alpha(\beta(g - y_{t-1}) - \dot{y}_{t-1} + f(x_t, g) \tag{4}$$

Through Euler integration, we can find the next velocity and position after a small time interval $dt$

$$\dot{y}_t = \dot{y}_{t-1} + \ddot{y}_{t-1}dt, \quad y_t = y_{t-1} + \dot{y}_{t-1}dt \tag{5}$$

In practice, this integration is implemented in $m$ discrete steps. To perform a forward pass, we unroll the integrator for $m$ iterations starting from initial $\dot{y}_0$, $\ddot{y}_0$. We can either apply all the $m$ intermediate robot states $y$ as actions on the robot using inverse controller $\Omega(.)$, or equally sub-sample them into $k \in \{1, m\}$ actions in between, where $k$ is the NDP rollout length. This frequency of sampling could allow robot operation at a much higher frequency (.5-5KHz) than the environment (usually 100Hz). The sampling frequency need not be same at training and inference as discussed further in Section 3.5.

Now we can compute gradients of the trajectory from the DMP with respect to $w$ and $g$ using Equations (3)-(5) as follows:

$$\frac{\partial f(x_t, g)}{\partial w_i} = \frac{\psi_i}{\sum_j \psi_j}(g - y_0)x_t, \quad \frac{\partial f(x_t, g)}{\partial g} = \frac{\psi_j w_j}{\sum_j \psi_j} x_t \tag{6}$$

Using this, a recursive relationship follows between, (similarly to the one derived by Pahic et al. [29]) $\frac{\partial y_t}{\partial w_i}$, $\frac{\partial y_t}{\partial g}$ and the preceding derivatives of $w_i$, $g$ with respect to $y_{t-1}$, $y_{t-2}$, $\dot{y}_{t-1}$ and $\dot{y}_{t-2}$. Complete derivation of equation (6) is given in appendix.

We now discuss how NDPs can be leveraged to train policies for imitation learning and reinforcement learning setups.

### 3.3 Training NDPs for Imitation (Supervised) Learning

Training NDPs in imitation learning setup is rather straightforward. Given a sequence of input $\{s, s', \dots\}$, NDP's $\pi(s; \theta)$ outputs a sequence of actions $a, a' \dots$. In our experiments, $s$ is a high dimensional image input. Let the demonstrated action sequence be $\tau_{\text{target}}$, we just take a loss between the predicted sequence as follows:

$$\mathcal{L}_{\text{imitation}} = \sum_s ||\pi(s) - \tau_{\text{target}}(s)||^2 \tag{7}$$

The gradients of this loss are backpropagated as described in Section 3.2 to train the parameters $\theta$.

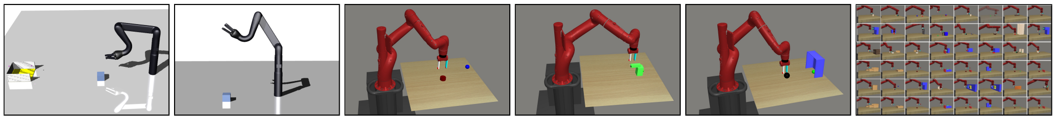
### 3.4 Training NDPs for Reinforcement Learning

We now show how an NDP can be used as a policy, $\pi$ in the RL setting. As discussed in Section 3.2, NDP samples k actions for the agent to execute in the environment given input observation $s$. One could use any underlying RL algorithm to optimize the expected future returns. In this paper, we use Proximal Policy Optimization (PPO) [42] and treat $a$ independently when computing the policy gradient for each step of the NDP rollout and backprop via a reinforce objective.

There are two choices for value function critic $V^\pi(s)$: either predict a single common value function for all the actions in the $k$-step rollout or predict different critic values for each step in the NDP rollout sequence. We found that the latter works better in practice. We call this a *multi-action critic architecture* and predict $k$ different estimates of value using $k$-heads on top of the critic network. Later, in the experiments we perform ablations over the choice of $k$. To further create a strong baseline comparison, as we discuss in Section 4, we also design and compare against a variant of PPO that predicts multiple actions using our multi-action critic architecture.

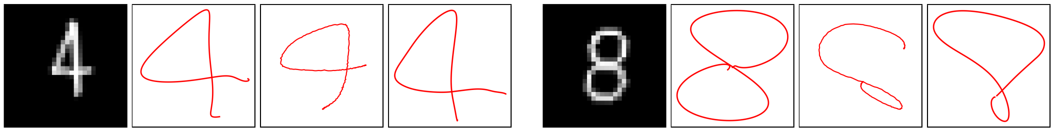
**Algorithm 1** Training NDPs for RL

```
Require: Policy $\pi$, $k$ NDP rollout length, $\Omega$ low-level inverse controller
  for $1, 2, ...$ episodes do
    for $t = 0, k, \dots,$ until end of episode do
      $w, g = \Phi(s_t)$
      Robot $y_t, \dot{y}_t$ from $s_t$ (pos, vel)
      for $m = 1, ..., M$ (integration steps) do
        Estimate $\dot{x}_m$ via (2) and update $x_m$
        Estimate $\ddot{y}_{t+m}, \dot{y}_{t+m}, y_{t+m}$ via (4), (5)
        $a = \Omega(y_{t+m}, y_{t+m-1})$
        Apply action $a$ to get $s'$
        Store transition $(s, a, s', r)$
      end for
      Compute Policy gradient $\nabla_\theta J$
      $\theta \leftarrow \theta + \eta \nabla_\theta J$
    end for
  end for
```



(a) Throwing (b) Picking (c) Pushing (d) Faucet Open (e) Soccer (f) 50 Tasks

Figure 3: Environment snapshot for different tasks considered in experiments. (a,b) Throwing and Picking tasks are adapted from [17] on the Kinova Jaco arm. (c-f) Remaining tasks are adapted from Yu et al. [50].



Input Ours CNN CNN-DMP Input Ours CNN CNN-DMP

Figure 4: Imitation (supervised) learning results on held-out test images of the digit writing task. Given an input image (left), the output action is the end-effector position of a planar robot. All methods have the same neural network architecture for fair comparison. We find that the trajectories predicted by NDPs (ours) are dynamically smooth as well as more accurate than both baselines (a vanilla CNN and the architecture from Pahic et al. [29].

Algorithm 1 provides a summary of our method for training NDPs with policy gradients. We only show results of using NDPs with on-policy RL (PPO), however, NDPs can similarly be adapted to off-policy methods.

### 3.5 Inference in NDPs

In the case of inference, our method uses the NDP policy $\pi$ once every $k$ environment steps, hence requires $k$-times fewer forward passes as actions applied to the robot. While reducing the inference time in simulated tasks may not show much difference, in real world settings, where large perception systems are usually involved, reducing inference time can help decrease overall time costs. Additionally, deployed real world systems may not have the same computational power as many systems used to train state-of-the-art RL methods on simulators, so inference costs end up accumulating, thus a method that does inference efficiently can be beneficial. Furthermore, as discussed in Section 3.2, the rollout length of NDP can be more densely sampled at test-time than at training allowing the robot to produce smooth and dynamically stable motions. Compared to about 100Hz frequency of the simulation, our method can make decisions an order of magnitude faster (at about 0.5-5KHz) at inference.

## 4 Experimental Setup

**Environments** To test our method on dynamic environments, we took existing torque control based environments for Picking and Throwing [17] and modified them to enable joint angle control. The robot is a 6-DoF Kinova Jaco Arm. In Throwing, the robot tosses a cube into a bin, and in Picking, the robot picks up a cube and lifts it as high as possible. To test on quasi-static tasks, we use Pushing, Soccer, Faucet-Opening from the Meta-World [50] task suite, as well as a setup that requires learning all 50 tasks (MT50) jointly (see Figure 3). These Meta-World environments are all in end-effector position control settings and based on a Sawyer Robot simulation in Mujoco [47]. In order to make the tasks more realistic, all environments have some degree of randomization. Picking and Throwing have random starting positions, while the rest have randomized goals.

**Baselines** We use PPO [42] as the underlying optimization algorithm for NDPs and all the other baselines compared in the reinforcement learning setup. The first baseline is the PPO algorithm itself without the embedded dynamical structure. Further, as mentioned in the Section 3.2, NDP is able to operate the robot at a much higher frequency than the world. Precisely, its frequency is $k$-times higher where $k$ is the NDP rollout length (described in Section 3.2). Even though the robot moves at a higher frequency, the environment/world state is only observed at normal rate, i.e., once every $k$ robot steps and the reward computation at the intermediate $k$ steps only uses the stale environment/world state from the first one of the $k$-steps. Hence, to create a stronger baseline that can also operate at higher frequency, we create a "PPO-multi" baseline that predicts multiple actions and also uses our *multi-action critic* architecture as described in Section 3.4. All methods are compared in terms of performance measured against the environment sample states observed by the agent. In addition, we also compare to Variable Impedance Control in End-Effector Space (VICES) [26] and Dynamics-Aware Embeddings (Dyn-E) [49] . VICES learns to output parameters of a PD controller or an Impedance controller directly. Dyn-E, on the other hand, using forward prediction based on environment dynamics, learns a lower dimensional action embedding.

## 5 Evaluation Results: NDPs for Imitation and Reinforcement Learning

We validate our approach on Imitation Learning and RL tasks in order to ascertain how NDP compares to state-of-the-art methods. We investigate: a) Does dynamical structure in NDPs help in learning from demonstrations in imitation learning setups?; b) How well do NDPs perform on dynamic and quasi-static tasks in deep reinforcement learning setups compared to the baselines?; c) How sensitive is the performance of NDPs to different hyper-parameter settings?

### 5.1 Imitation (Supervised) Learning

To evaluate NDPs in imitation learning settings we train an agent to perform various control tasks. We evaluate NDPs on the Mujoco [47] environments discussed in Section 4 (Throwing, Picking, Pushing, Soccer and Faucet-Opening). Experts are trained using PPO [42] and are subsequently used to collect trajectories. We train an NDP via the behaviour cloning procedure described in Section 3.3, on the collected expert data. We compare against a neural network policy (using roughly the same model capacity for both). Success rates in Table 1 indicate that NDPs show superior performance on a wide variety of control tasks.

| Method | NN | **NDP (ours)** |
|---|---|---|
| Throw | $0.528 \pm 0.262$ | $\mathbf{0.642 \pm 0.246}$ |
| Pick | $\mathbf{0.672 \pm 0.074}$ | $0.408 \pm 0.058$ |
| Push | $0.002 \pm 0.004$ | $\mathbf{0.208 \pm 0.049}$ |
| Soccer | $0.885 \pm 0.016$ | $\mathbf{0.890 \pm 0.010}$ |
| Faucet | $0.532 \pm 0.231$ | $\mathbf{0.790 \pm 0.059}$ |

Table 1: Imitation (supervised) learning results (success rates between 0 and 1) on Mujoco [47] environments. We see that NDP outperforms the neural network baseline in many tasks.

In order to evaluate the ability of NDPs to handle complex visual data, we perform a digit-writing task using a 2D end-effector. The goal is to train a planar robot to trace the digit, given its image as input. The output action is the robot's end-effector position, and supervision is obtained by treating ground truth trajectories as demonstrations. We compare NDPs to a regular behavior cloning policy parameterized by a CNN and the prior approach which maps image to DMP parameters [29] (dubbed, CNN-DMP). CNN-DMP [29] trains a single DMP for the whole trajectory and requires supervised demonstrations. In contrast, to NDPs can generate multiple DMPs across time and can be used in an RL setup as well. However, for a fair comparison, we compare both methods apples-to-apples with single DMP for whole trajectory, i.e., $k = 300$.

Qualitative examples are in Figure 4, and quantitative results in Table 2 report the mean loss between output trajectory and ground truth. NDP outperforms both CNN and CNN-DMP [29] drastically. Our method also produces much higher quality and smoother reconstructions as shown in Figure 4. Results show that our method can efficiently capture dynamic motions in a supervised setting, while learning from visual data.

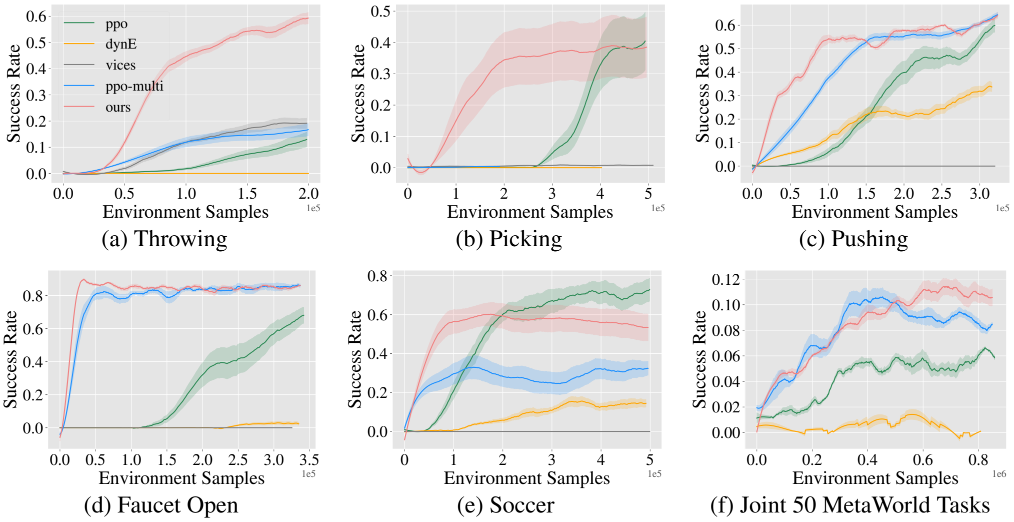
| Method | Train | Test (held-out) |
|---|---|---|
| CNN | $10.42 \pm 5.26$ | $10.59 \pm 4.63$ |
| CNN-DMP [29] | $9.44 \pm 4.59$ | $8.46 \pm 8.45$ |
| NDP (ours) | $\mathbf{0.70 \pm 0.36}$ | $\mathbf{0.74 \pm 0.34}$ |

Table 2: Imitation learning on digit writing task. We report the mean loss across 10 digit classes. The input is the image of the digit to be written and action output is the end-effector position of robot. Our method significantly outperforms the baseline.

### 5.2 Reinforcement Learning

In contrast to imitation learning where the rollout length of NDP is high ($k = 300$), we set $k = 5$ in RL because the reward becomes too sparse if $k$ is very large. We compare the success rate of our method with that of the baseline methods PPO, PPO-multi with $k = 5$, VICES and DYN-E.

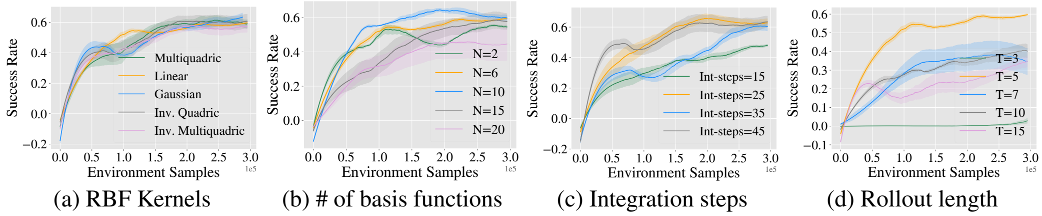
As shown in In Figure 5, our method NDP sees gains in both efficiency and performance in most tasks. In Soccer, PPO reaches a higher final performance, but our method shows twice the efficiency at a small loss in performance. The final task of training jointly across 50 Meta-World tasks is too hard for all methods. Nevertheless, our NDP attains a slightly higher absolute performance than baseline but doesn't show efficiency gains over baselines.



(a) Throwing (b) Picking (c) Pushing

(d) Faucet Open (e) Soccer (f) Joint 50 MetaWorld Tasks

Figure 5: Evaluation of reinforcement learning setup for continuous control tasks. Y axis is success rate and X axis is number of environment samples. We compare to PPO [42], a multi-action version of PPO, VICES [26] and DYN-E [49]. The dynamic rollout for NDP & PPO-multi is $k = 5$.



(a) RBF Kernels (b) # of basis functions (c) Integration steps (d) Rollout length

Figure 6: Ablation of NDPs with respect to different hyperparameters in the RL setup (pushing). We ablate different choices of radial basis functions in (a). We ablate across number of basis functions, integration steps, and length of the NDP rollout in (b,c,d). Plots indicate that NDPs are fairly stable across a wide range of choices.

PPO-multi, a multi-action algorithm based on our proposed multi-action critic setup tends to perform well in some case (Faucet Opening, Pushing etc) but is inconsistent in its performance across all tasks and fails completely at times, (Picking etc.). Our method also outperforms prior state-of-the-art methods that re-paremeterize action spaces, namely, VICES [26] and Dyn-E [49]. VICES is only slightly successful in tasks like throwing, since a PD controller can efficiently solve the task, but suffer in more complex settings due to a large action space dimensionality (as it predicts multiple quantities per degree of freedom). Dyn-E, on the other hand, performs well on tasks such as Pushing, or Soccer, which have simpler dynamics and contacts, but fails to scale to more complex environments.

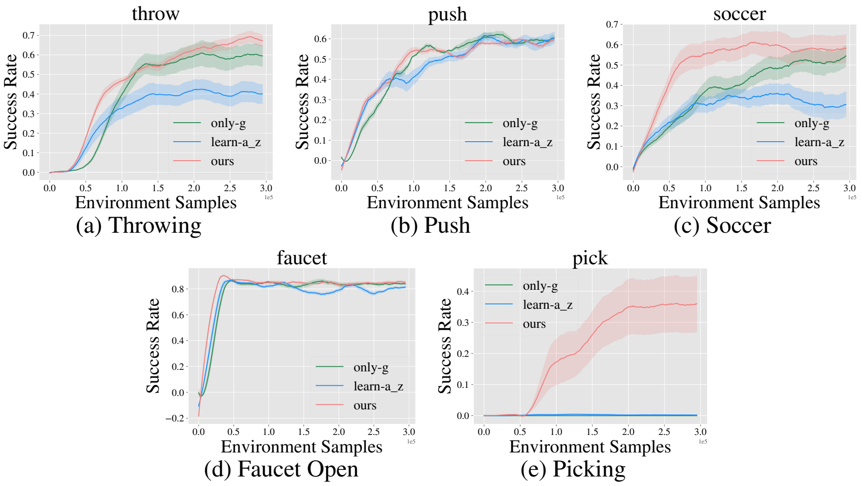
Through these experiments, we show the diversity and versatility of NDP, as it has a strong performance across different types of control tasks. NDP outperforms baselines in both dynamic (throwing) and static tasks (pushing) while being able to learn in a more data efficient manner. It is able to reason in a space of physically meaningful trajectories, but it does not lose the advantages and flexibility that other policy setups have.

### 5.2.1 Ablations for NDPs in Reinforcement Learning Setup

We aim to understand how design choices affect the RL performance of NDP. We run comparisons on the pushing task, varying the number of basis functions $N$ (in the set $\{2, 6, 10, 15, 20\}$), DMP rollout lengths (in set $\{3, 5, 7, 10, 15\}$), number of integration steps (in set $\{15, 25, 35, 45\}$), as well as different basis functions: Gaussian RBF (standard), $\psi$ defined in Equation (3), a liner map $\psi(x) = x$, a multiquadric map: $\psi(x) = \sqrt{1 + (\epsilon x)^2}$, a inverse quadric map $\psi(x) = \frac{1}{1+(\epsilon x)^2}$, and an inverse multiquadric map: $\psi(x) = \frac{1}{\sqrt{1+(\epsilon x)^2}}$.

Additionally, we investigate the effect of different NDP components on its performance. To this end, we ablate a setting where only $g$ (the goal) is learnt while the radial basis function weights (the forcing function) are 0 (we call this setting 'only-g'). We also ablate a version of NDP that learns the global constant $\alpha$ (from Equation 4), in addition to the other parameters ($g$ and $w$).

Figure 6 shows results from ablating different NDP parameters. Varying $N$ (number of basis functions) controls the shape of the trajectory taken by the agent. A small $N$ may not have the power to represent the nuances of the motion required, while a big $N$ may make the parameter space too large to learn efficiently. We see that number of integration steps do not have a large effect on performance, similarly to the type of radial basis function. Most radial basis functions generally have similar interpolation and representation abilities. We see that $k = 3$ (the length of each individual rollout within NDP) has a much lower performance due to the fact that 3 steps cannot capture the smoothness or intricacies of a trajectory. Overall, we mostly find that NDP is robust to design choices. Figure 7 shows that the current formulation of NDP outperforms the one where $\alpha$ is learnt. We also observe that setting the forcing term to 0 (only learning the goal, $g$) is significantly less sample efficient than NDPs while converging to a slightly lower asymptotic performance.



(a) Throwing (b) Push (c) Soccer

(d) Faucet Open (e) Picking

Figure 7: Ablations for different NDP design choices. The first entails NDP also learning the parameter $\alpha$ (shown as $a_z$). In the second one, $g$ is learnt but not $w_i$, i.e. the forcing function is 0 ('only-g'). Results indicate that NDP outperforms both these settings.

## 6 Related Work

**Dynamic Movement Primitives** Previous works have proposed and used Dynamic Movement Primitives (DMP) [22, 36, 41] for robot control [24, 27, 35]. Work has been done in representing dynamical systems, both as extensions of DMPs [6, 7, 11, 48], and beyond DMPs by learning kernels [19] and designing Riemannian metrics [39]. Learning methods have been used to incorporate environment sensory information into the forcing function of DMPs [37, 45]. DMPs have also been prime candidates to represent primitives when learning hierarchical policies, given the range of motions DMPs can be used for in robotics [13, 23, 32, 44]. Parametrization of DMPs using gaussian processes has also been proposed to facilitate generalization [34, 48]. Recently, deep learning has also been used to map images to DMPs [29] and to learn DMPs in latent feature space [8]. However most of these works require pre-trained DMPs via expert demonstrations or are only evaluated in the supervised setting. Furthermore, either a single DMP is used to represent the whole trajectory or the demonstration is manually segmented to learn a different DMP for each segment. In contrast, NDPs outputs a new dynamical system for each timestep to fit diverse trajectory behaviours across time. Since we embed dynamical structure into the deep network, NDPs can flexibly be incorporated not just in visual imitation but also in deep reinforcement learning setups, in an end-to-end manner.

**Reparameterized Policy and Action Spaces** A broader area of work that makes use of action reparameterization is the study of Hierarchical Reinforcement Learning (HRL). Works in the options framework [4, 46] attempt to learn an overarching policy that controls usage of lower-level policies or primitives. Lower-level policies are usually pre-trained therefore require supervision and knowledge of the task beforehand, limiting the generalizability of such methods. For example, Daniel et al. [13], Parisi et al. [30] incorporate DMPs into option-based RL policies, using a pre-trained DMPs as the high level actions. This setup requires re-learning DMPs for different types of tasks and does not allow the same policy to generalize, since it needs to have access to an extremely large number of DMPs. Action spaces can also be reparameterized in terms of pre-determined PD controller [51] or learned impedance controller parameters [26]. While this helps for policies to adapt to contact rich behaviors, it does not change the trajectories taken by the robot. This often leads to high dimensionality, and thus a decrease sample efficiency. In addition, Whitney et al. [49] learn an action embedding based on passive data, however, this does not take environment dynamics or explicit control structure into account.

**Structure in Policy Learning** Various methods in the field of control and robotics have employed physical knowledge, dynamical systems, optimization, and more general task/environment dynamics to create structured learning. Works such as [12, 18] propose networks constrained through physical properties such as Hamiltonian co-ordinates or Lagrangian Dynamics. However, the scope of these works is limited to toy examples such as a point mass, and are often used for supervised learning. Similarly, other works [28, 33, 38, 40] all employ dynamical systems to model demonstrations, and do not tackle generalization or learning beyond imitation. Fully differentiable optimization problems have also been incorporated as layers inside a deep learning setup [1, 2, 9]. Whil they share the underlying idea of embedding structure in deep networks such that some aspects of this structure can be learned end-to-end, they have not been explored in tackling complex robotic control tasks. Furthermore, it is common in RL setups to incorporate planning based on a system model [3, 10, 14–16]. However, this is usually learned from experience or from attempts to predict the effects of actions on the environment (forward and inverse models), and often tends to fail for complex dynamic tasks.

## 7 Discussion

Our method attempts to bridge the gap between classical robotics, control and recent approaches in deep learning and deep RL. We propose a novel re-parameterization of action spaces via Neural Dynamic Policies, a set of policies which impose the structure of a dynamical system on action spaces. We show how this set of policies can be useful for continuous control with RL, as well as in supervised learning settings. Our method obtains superior results due to its natural imposition of structure and yet it is still generalizable to almost any continuous control environment.

The use of DMPs in this work was a particular design choice within our architecture which allows for any form of dynamical structure that is differentiable. As alluded to in the introduction, other similar representations can be employed in their place. In fact, DMPs are a special case of a general second order dynamical system [5, 39] where the inertia term is identity, and potential and damping functions are defined in a particular manner via first order differential equations with a separate forcing function which captures the complexities of the desired behavior. Given this, one can setup a dynamical structure such that it explicitly models and learns the metric, potential, and damping explicitly. While this brings advantages in better representation, it also brings challenges in learning. We leave these directions for future work to explore.

## Acknowledgments

We thank Giovanni Sutanto, Stas Tiomkin and Adithya Murali for fruitful discussions. We also thank Franziska Meier, Akshara Rai, David Held, Mengtian Li, George Cazenavette, and Wen-Hsuan Chu for comments on early drafts of this paper. This work was supported in part by DARPA Machine Common Sense grant and Google Faculty Award to DP.

## References

[1] B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *ICML*, 2017. 9

[2] B. Amos, I. D. J. Rodriguez, J. Sacks, B. Boots, and J. Z. Kolter. Differentiable mpc for end-to-end planning and control. In *NeurIPS*, 2018. 9

[3] C. G. Atkeson and J. C. Santamaria. A comparison of direct and model-based reinforcement learning. In *ICRA*, 1997. 9

[4] P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *AAAI*, 2017. 8

[5] F. Bullo and A. D. Lewis. *Geometric Control of Mechanical Systems*. Springer, 2005. 9

[6] S. Calinon. A tutorial on task-parameterized movement learning and retrieval. *Intelligent Service Robotics*, 2016. 8

[7] S. Calinon, I. Sardellitti, and D. G. Caldwell. Learning-based control strategy for safe human-robot interaction exploiting task and robot redundancies. *IROS*, 2010. 8

[8] N. Chen, M. Karl, and P. Van Der Smagt. Dynamic movement primitives in latent space of time-dependent variational autoencoders. In *International Conference on Humanoid Robots (Humanoids)*, 2016. 8

[9] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. In *NeurIPS*, 2018. 9

[10] K. Chua, R. Calandra, R. McAllister, and S. Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *arXiv preprint arXiv:1805.12114*, 2018. 9

[11] A. Conkey and T. Hermans. Active learning of probabilistic movement primitives. *2019 IEEE-RAS 19th International Conference on Humanoid Robots (Humanoids)*, 2019. 8

[12] M. Cranmer, S. Greydanus, S. Hoyer, P. Battaglia, D. Spergel, and S. Ho. Lagrangian neural networks. *arXiv preprint arXiv:2003.04630*, 2020. 9

[13] C. Daniel, G. Neumann, O. Kroemer, and J. Peters. Hierarchical relative entropy policy search. *Journal of Machine Learning Research*, 2016. 8, 9

[14] M. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *ICML*, 2011. 9

[15] M. P. Deisenroth, G. Neumann, and J. Peters. A survey on policy search for robotics. *Found. Trends Robot*, 2013.

[16] M. P. Deisenroth, D. Fox, and C. E. Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2015. 9

[17] D. Ghosh, A. Singh, A. Rajeswaran, V. Kumar, and S. Levine. Divide-and-conquer reinforcement learning. *arXiv preprint arXiv:1711.09874*, 2017. 5, 15

[18] S. Greydanus, M. Dzamba, and J. Yosinski. Hamiltonian neural networks. In *NeurIPS*, 2019. 9

[19] Y. Huang, L. Rozo, J. Silvério, and D. G. Caldwell. Kernelized movement primitives. *The International Journal of Robotics Research*, 2019. 8

[20] A. J. Ijspeert, J. Nakanishi, and S. Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *ICRA*. IEEE, 2002. 2

[21] A. J. Ijspeert, J. Nakanishi, and S. Schaal. Learning attractor landscapes for learning motor primitives. In *NeurIPS*, 2003.

[22] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal. Dynamical movement primitives: Learning attractor models for motor behaviors. *Neural Computation*, 2013. 2, 3, 8

[23] J. Kober and J. Peters. Learning motor primitives for robotics. In *ICRA*, 2009. 8

[24] P. Kormushev, S. Calinon, and D. G. Caldwell. Robot motor skill coordination with em-based reinforcement learning. In *IROS*, 2010. 8

[25] I. Kostrikov. Pytorch implementations of reinforcement learning algorithms. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail, 2018. 15, 16

[26] R. Martin-Martin, M. A. Lee, R. Gardner, S. Savarese, J. Bohg, and A. Garg. Variable impedance control in end-effector space: An action space for reinforcement learning in contact-rich tasks. *IROS*, 2019. 6, 7, 9, 15, 16

[27] K. Mülling, J. Kober, O. Kroemer, and J. Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 2013. 8

[28] K. Neumann and J. Steil. Learning robot motions with stable dynamical systems under diffeomorphic transformations. *Robotics and Autonomous Systems*, 2015. 9

[29] R. Pahic, A. Gams, A. Ude, and J. Morimoto. Deep encoder-decoder networks for mapping raw images to dynamic movement primitives. *ICRA*, 2018. 4, 5, 6, 8

[30] S. Parisi, H. Abdulsamad, A. Paraschos, C. Daniel, and J. Peters. Reinforcement learning vs human programming in tetherball robot games. In *IROS*, 2015. 9

[31] P. Pastor, H. Hoffmann, T. Asfour, and S. Schaal. Learning and generalization of motor skills by learning from demonstration. In *ICRA*, 2009. 3

[32] P. Pastor, M. Kalakrishnan, S. Chitta, E. Theodorou, and S. Schaal. Skill learning and task outcome prediction for manipulation. In *ICRA*, 2011. 8

[33] N. Perrin and P. Schlehuber-Caissier. Fast diffeomorphic matching to learn globally asymptotically stable nonlinear dynamical systems. *Systems & Control Letters*, 2016. 9

[34] A. Pervez and D. Lee. Learning task-parameterized dynamic movement primitives using mixture of gmms. *Intelligent Service Robotics*, 2018. 8

[35] J. Peters, S. Vijayakumar, and S. Schaal. Reinforcement learning for humanoid robotics. In *International Conference on Humanoid Robots*, 2003. 8

[36] M. Prada, A. Remazeilles, A. Koene, and S. Endo. Dynamic movement primitives for human-robot interaction: Comparison with human behavioral observation. In *International Conference on Intelligent Robots and Systems*, 2013. 8

[37] A. Rai, G. Sutanto, S. Schaal, and F. Meier. Learning feedback terms for reactive planning and control. In *ICRA*, 2017. 8

[38] M. A. Rana, A. Li, D. Fox, B. Boots, F. Ramos, and N. Ratliff. Euclideanizing flows: Diffeomorphic reduction for learning stable dynamical systems. *arXiv preprint arXiv:2005.13143*, 2020. 9

[39] N. D. Ratliff, J. Issac, D. Kappler, S. Birchfield, and D. Fox. Riemannian motion policies. *arXiv preprint arXiv:1801.02854*, 2018. 2, 8, 9

[40] H. Ravichandar, I. Salehi, and A. Dani. Learning partially contracting dynamical systems from demonstrations. In *CoRL*, 2017. 9

[41] S. Schaal. Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In *Adaptive motion of animals and machines*. Springer, 2006. 2, 3, 8

[42] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017. 4, 5, 6, 7, 15, 16

[43] M. W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot modeling and control*. John Wiley & Sons, 2020. 2

[44] F. Stulp, E. A. Theodorou, and S. Schaal. Reinforcement learning with sequences of motion primitives for robust manipulation. *Transactions on Robotics*, 2012. 8

[45] G. Sutanto, Z. Su, S. Schaal, and F. Meier. Learning sensor feedback models from demonstrations via phase-modulated neural networks. In *ICRA*, 2018. 8

[46] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 1999. 8

[47] E. Todorov, T. Erez, and Y. Tassa. MuJoCo: A physics engine for model-based control. In *The IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012. 5, 6

[48] A. Ude, A. Gams, T. Asfour, and J. Morimoto. Task-specific generalization of discrete and periodic dynamic movement primitives. *Transactions on Robotics*, 2010. 8

[49] W. Whitney, R. Agarwal, K. Cho, and A. Gupta. Dynamics-aware embeddings. *arXiv preprint arXiv:1908.09357*, 2019. 6, 7, 9, 15, 16

[50] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. *arXiv preprint arXiv:1910.10897*, 2019. 5, 15

[51] Y. Yuan and K. Kitani. Ego-pose estimation and forecasting as real-time pd control. In *ICCV*, 2019. 9

# A Appendix

## A.1 Videos

Videos can also be found at: https://shikharbahl.github.io/neural-dynamic-policies/. We found that NDP results look dynamically more stable and smooth in comparison to the baselines. PPO-multi generates shaky trajectories, while corresponding NDP (ours) trajectories are much smoother .This is perhaps due to the embedded dynamical structure in NDPs as all other aspects in PPO-multi and NDP (ours) are compared apples-to-apples.

## A.2 Differentiability Proof of Dynamical Structure in NDPs

In Section 3.2, we provide an intuition for how NDP is incorporates a second order dynamical system (based on the DMP system, described in Section 2) in a differentiable manner. Let us start by observing that, when implementing our algorithm, $y_0$, $\dot{y}_0$ are known and $\ddot{y}_0 = 0$, as well as $x_0 = 1$. Assuming that the output states of NDP are $y_0, y_1, ..., y_t, ...$ and assuming that there exists a loss function $L$ which takes in $y_t$, we want partial derivatives with respect to DMP weights $w_i$ and goal $g$:

$$\frac{\partial L(y_t)}{\partial w_i}, \quad \frac{\partial L(y_t)}{\partial w_i} \tag{8}$$

$$\frac{\partial L(y_t)}{\partial y_t}\frac{\partial y_t}{\partial w_i} \tag{9}$$

Starting with $w_i$, using the Chain Rule we get that

$$\frac{\partial L(y_t)}{\partial w_i} = \frac{\partial L(y_t)}{\partial y_t}\frac{\partial y_t}{\partial w_i} \tag{10}$$

Hence, we want to be able to calculated $\frac{\partial y_t}{\partial w_i}$. For simplicity let:

$$W_t = \frac{\partial y_t}{\partial w_i} \tag{11}$$

$$\dot{W}_t = \frac{\partial \dot{y}_t}{\partial w_i} \tag{12}$$

$$\ddot{W}_t = \frac{\partial \ddot{y}_t}{\partial w_i} \tag{13}$$

From section 3.2 we know that:

$$\ddot{y}_t = \alpha(\beta(g - y_{t-1}) - \dot{y}_{t-1} + f(x_t, g) \tag{14}$$

and, the discretization over a small time interval $dt$ gives us:

$$\dot{y}_t = \dot{y}_{t-1} + \ddot{y}_{t-1}dt, \quad y_t = y_{t-1} + \dot{y}_{t-1}dt \tag{15}$$

From these and the fact that $y_0$, $\dot{y}_0$ are known and $\ddot{y}_0 = 0$, as well as $x_0 = 1$, we get that $y_1 = y_0 + \dot{y}_0 dt$ and $\dot{y}_1 = \dot{y}_0 + 0dt = \dot{y}_0$, as well as $\ddot{y}_1 = \alpha(\beta(g - y_0) - \dot{y}_0 + f(x_1, g)$.

Using Equations (14) and (15) we get that:

$$W_t = \frac{\partial}{\partial w_i}(y_{t-1} + \dot{y}_{t-1}dt) \tag{16}$$

$$W_t = W_{t-1} + \dot{W}_{t-1}dt \tag{17}$$

and

$$\dot{W}_{t-1} = \dot{W}_{t-2} + \ddot{W}_{t-1}dt \tag{18}$$

In turn,

$$\ddot{W}_{t-1} = \frac{\partial}{\partial w_i}(\alpha(\beta(g - y_{t-2}) - \dot{y}_{t-2}) + f(x_{t-1}, g)) \tag{19}$$

From section 3.2, we know that

$$\frac{\partial f(x_{t-1}, g)}{\partial w_i} = \frac{\psi_i}{\sum_j \psi_j}(g - y_0)x_{t-1} \tag{20}$$

Hence:

$$\ddot{W}_{t-1} = \alpha(\beta(-W_{t-2}) - \dot{W}_{t-2}) + \frac{\psi_i}{\sum_j \psi_j}(g - y_0)x_{t-1} \tag{21}$$

Plugging equations the value of $\ddot{W}_{t-1}$ into Equation (18):

$$\dot{W}_{t-1} = \dot{W}_{t-2} + (\alpha(\beta(-W_{t-2}) - \dot{W}_{t-2}) + \frac{\psi_i}{\sum_j \psi_j}(g - y_0)x_{t-1})dt \tag{22}$$

Now plugging the value of $\dot{W}_{t-1}$ in Equation (17):

$$W_t = W_{t-1} + (\dot{W}_{t-2} + (\alpha(\beta(-W_{t-2}) - \dot{W}_{t-2}) + \frac{\psi_i}{\sum_j \psi_j}(g - y_0)x_{t-1})dt)dt \tag{23}$$

We see that the value of $W_t$ is dependent on $W_{t-1}, \dot{W}_{t-2}, W_{t-2}$. We can now show that we can acquire a numerical value for $W_t$ by recursively following the gradients, given that $W_{t-1}, \dot{W}_{t-2}, W_{t-2}$ are known. Since we showed that $y_0$, $\dot{y}_0$, $y_1$,$\dot{y}_1$ do not require $w_i$ in their computation, $W_1, \dot{W}_0, W_0 = 0$. Hence by recursively following the relationship defined in Equation (23), we achieve a solution for $W_t$.

Similarly, let:

$$G_t = \frac{\partial}{\partial g}(y_{t-1} + \dot{y}_{t-1}dt) \tag{24}$$

$$G_t = G_{t-1} + \dot{G}_{t-1}dt \tag{25}$$

and

$$\dot{G}_{t-1} = \dot{G}_{t-2} + \ddot{G}_{t-1}dt \tag{26}$$

Using section 3.2, we get that

$$\frac{\partial f(x_{t-1}, g)}{\partial g} = \frac{\psi_j w_j}{\sum_j \psi_j}x_{t-1} \tag{27}$$

Hence:

$$\ddot{G}_{t-1} = \alpha(\beta(1 - G_{t-2}) - \dot{G}_{t-2}) + \frac{\psi_j w_j}{\sum_j \psi_j}x_{t-1} \tag{28}$$

and we get a similar relationship as Equation (23):

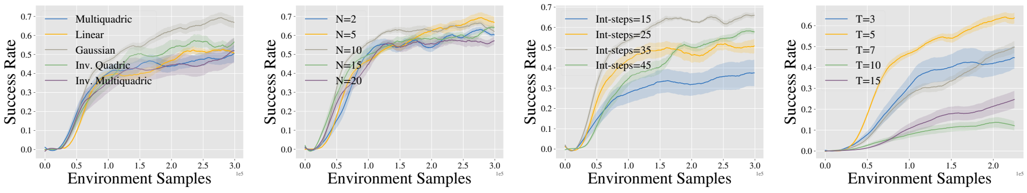
$$G_t = G_{t-1} + (\dot{G}_{t-2} + (\alpha(\beta(1 - G_{t-2}) - \dot{G}_{t-2}) + \frac{\psi_j w_j}{\sum_j \psi_j}x_{t-1})dt)dt \tag{29}$$

Hence, $G_t$, similarly is dependent on $G_{t-1}, \dot{G}_{t-2}, G_{t-2}$. We can use a similar argument as with $w_i$ to show that $G_t$ is also numerically achievable. We have now shown that $y_t$, the output of the dynamical system defined by a DMP, is differentiable with respect to $w_i$ and $g$.

### A.3 Ablations

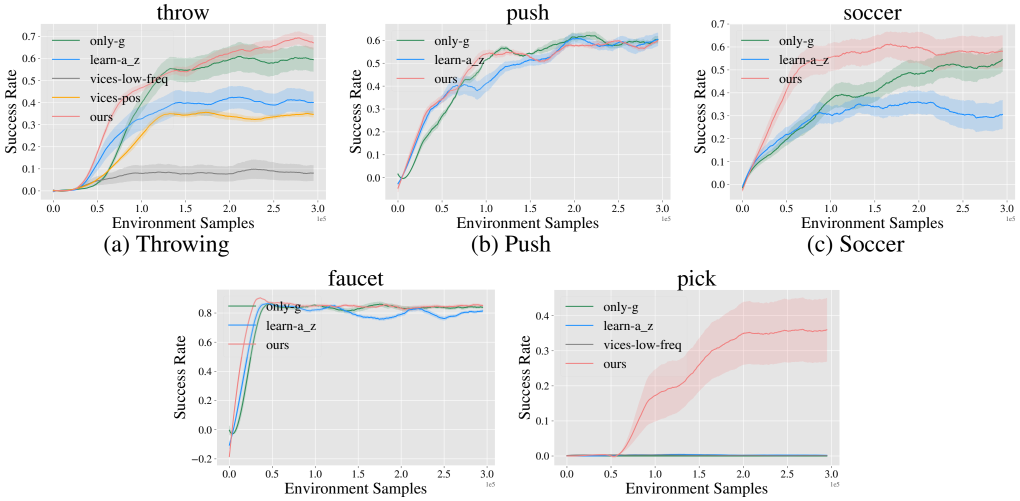
We present ablations similar to the ones in our paper, using the Throwing task. The results are showin in Figure 8. We see that NDPs show similar robustness across all variations. Secondly, we ran ablations with the forcing term set to 0 and found it variant to be significantly less sample efficient than NDPs while converging to a slightly lower asymptotic performance. Finally, we ran ablation where $\alpha$ is also learned by the policy while setting $\beta = \frac{\alpha}{4}$ for critical damping. We see in Figure 9 that NDPs outperforms both settings where $\alpha$ is learnt and where the the forcing term $f$ is set to 0.

Additionally, we ran multiple ablations for the VICES baseline. We present a version of VICES for throwing and picking tasks that acts in end-effector space instead of joint-space (we call this ‘vices-pos‘)., as well e ran another version of VICES where the higher level policy runs at similar frequency as NDP which we call ‘vices-low-freq‘. The results are presented in Figure 9a and Figure 9b. We found it to be less sample efficient and have a lower performance than NDP.



(a) RBF Kernels (b) # of basis functions (c) Integration steps (d) Rollout length

Figure 8: Ablation of NDPs with respect to different hyperparameters in the RL setup (throw). We ablate different choices of radial basis functions in (a). We ablate across number of basis functions, integration steps, and length of the NDP rollout in (b,c,d). Plots indicate that NDPs are fairly stable across a wide range of choices.



(a) Throwing (b) Push (c) Soccer

(d) Faucet Open (e) Picking

Figure 9: Ablations for learning $\alpha$ (az) as well, only learning $g$ (only-g), VICES at a lower control frequency (vices-lower-freq)and VICES for Throw in end-effector space (vices-pos)

## A.4 Implementation Details

### Hyper-parameters and Design Choices

We use default parameters and stick as closely as possible to the default code. In multi-action cases (for PPO-multi and NDP), we kept rollout length $k$ fixed for training and at inference time, one can sample arbitrarily high value for $k$ as demanded by the task setup. For reinforcement learning, we kept $k = 5$ because the reward becomes too sparse if $k$ is very large. For imitation learning, this is not an issue, and hence, $k = 300$ for learning from demonstration.

In reinforcement learning setup for NDP, we tried number of basis functions in [4, 5, 6, 7, 10] for each RL task. We fixed the number of integration steps per NDP rollout to 35. We also tried $\alpha$ (as described in Section 2) values in [10, 15, 25]. NDP (ours), PPO [42], PPO-Multi, VICES [26] all use a similar 3-layer fully-connected network of hidden layer sizes of [100, 100] with tanh non-linearities. All these use PPO [42] as the underlying RL optimizer. For Dyn-E [49], we used off-the-shelf architecture because it is based on off-policy RL and doesn't use PPO.

Hyper-parameters for underlying PPO optimization use off-the-shelf without any further tuning from PPO [42] implementation in Kostrikov [25] as follows:

### Environments

For Picking and Throwing, we adapted tasks from Ghosh et al. [17] (`https://github.com/dibyaghosh/dnc`. We modified these tasks to enable joint-angle position control. For other RL tasks, we used the Meta-World environment package [50] (`https://github.com/rlworkgroup/metaworld`). Since VICES [26] operates using torque control, we modified Meta-World environments to support torque-based Impedance control.

| **Hyperparameter** | **Value** |
|---|---|
| Learning Rate | $3 \times 10^{-4}$ |
| Discount Factor | 0.99 |
| Use GAE | True |
| GAE Discount Factor | 0.95 |
| Entropy Coefficient | 0 |
| Normalized Observations | True |
| Normalized Returns | True |
| Value Loss Coefficient | 0.5 |
| Maximum Gradient Norm | 0.5 |
| PPO Mini-Batches | 32 |
| PPO Epochs | 10 |
| Clip Parameter | 0.1 |
| Optimizer | Adam |
| Batch Size | 2048 |
| RMSprop optimizer epsilon | $10^{-5}$ |

Further, as mentioned in the Section 3.4 and 4, NDP and PPO-multi are able to operate the robot at a higher frequency than the world. Precisely, frequency is $k$-times higher where $k = 5$ is the NDP rollout length (described in Section 3.2). Even though the robot moves at higher frequency, the environment/world state is only observed at normal rate, i.e., once every $k$ robot steps and the reward computation at the intermediate $k$ steps only use stale environment/world state from the first one of the $k$-steps. For instance, if the robot is pushing a puck, the reward is function of robot as well as puck's location. The robot will knows its own position at every policy step but will have access to stale value of puck's location only from actual environment step (sampled at a lower frequency than policy steps, specifically 5x less). We implemented this for all 50 Meta-World environments as well as Throwing and Picking.

**Codebases: NDPs (ours) and Baselines**

Our code can be found at: `https://shikharbahl.github.io/neural-dynamic-policies/`. Our algorithm is based on top of Proximal Policy Optimization (PPO) [42] from `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail` [25]. Additionally, we use code from Whitney et al. [49] (DYN-E): `https://github.com/willwhitney/dynamics-aware-embeddings`. For our implementation of VICES [26], we use the controllers provided them in `https://github.com/pairlab/robosuite/tree/vices_iros19/robosuite` and overlay those on our environments.